UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Framework para diseño de diagramas de bloques funcionales para procesamiento de datos en tiempo real
Autores: Odetti, Alessandro
Director: Naiouf, Marcelo
Carrera: Licenciatura en Sistemas

Resumen

El trabajo propone y desarrolla un framework que integra soluciones de simulación y de programación visual, una herramienta que al ser integrada permite generar interfaces útiles para llevar a cabo la construcción de diagramas de flujos de datos, donde estos datos son alterados/procesados a través de diferentes bloques funcionales en tiempo real.

La herramienta provee un conjunto inicial de bloques con operaciones y comportamientos simples o básicos, y proporciona sencillos métodos que permiten a los desarrolladores generar diferentes tipos de bloques que sean requeridos para las necesidades del sistema en el cual se encuentra embebida.

El framework es capaz de resolver diversos problemas en diferentes ámbitos, donde el usuario final puede alterar la lógica de un sistema realizando cambios en el procesamiento de los datos de entrada, sin intervención de desarrolladores o programadores especializados.

Palabras Claves

Framework, bloques funcionales, diagramas de bloques, programación visual, flujos de datos, diagramas de datos, simulación, Flowgramming

Trabajos Realizados

En primera instancia se ha realizado una investigación de herramientas similares a la propuesta, así como de los diferentes tópicos principales relacionados con el objetivo, de modo de poder analizar el estado actual y las ventajas y desventajas de las alternativas presentes en la actualidad.
Como paso posterior, se ha desarrollado el framework planteado teniendo en cuenta la investigación previa, de modo de asegurar la obtención de una alternativa superadora a las herramientas presentes.

Conclusiones

El objetivo principal del trabajo de desarrollar una herramienta capaz de procesar datos en tiempo real a través de la generación de diagramas de flujos de datos ha sido cumplido, resultando en un framework que fusiona características de simulación y de desarrollo. La herramienta es a la vez flexible, de modo que, a través de diferentes posibles configuraciones, puede ser ajustada para funcionar colaborativa/sinérgicamente con otros sistemas para alterar la lógica de los mismos a través de diagramas de flujos de datos.

Trabajos Futuros

Se proponen como trabajos futuros una serie de modificaciones / extensiones con el objetivo de proveer una herramienta más completa y universal, de modo que pueda ser utilizada en una mayor gama de casos y diferentes entornos, así como también para obtener mejoras de eficiencia y escalabilidad.

Fecha de la presentación: Febrero 2018

Agradecimientos

Quiero expresar en estas breves líneas, mis más sinceros agradecimientos a todas aquellas personas que han hecho posible, de forma directa o indirecta, la realización de ese trabajo.

En primer lugar, agradezco a la UNLP en general, tanto a nivel escolar como universitario, por ser un excelente ámbito de estudios, en particular a mis profesores de "El Liceo" y de la Facultad de Informática.

Especiales agradecimientos al Dr. Marcelo Naiouf por aceptar y dirigir de forma más que predispuesta y atenta este trabajo.

Particulares agradecimientos para toda mi familia, y en especial a mi abuela Mirta por su amor y ternura incondicional. A todos mis amigos. Gracias por su apoyo y por permitirme compartir infinidad de momentos a su lado, sin ustedes sería otra persona.

Aprovecho esta especial oportunidad para agradecer por sobre todas las cosas a mi mamá. Mencionar todo lo brindado sería infinito y es por eso que me voy a limitar a unas pocas palabras:

> Ma, te agradezco no solo por el apoyo para concluir este trabajo y también esta importante etapa de la vida, sino por la excelente forma en la que cumpliste y cumplís con el rol de madre, y muchas veces más que eso. Por ser la persona que ha sabido mostrar indiscutiblemente el camino correcto a seguir frente a las diferentes situaciones de la vida, lejos de ser el camino correcto el más sencillo o el más cómodo. Con Sol y Luciana estamos y estaremos eternamente agradecidos y orgullosos de la madre que nos ha tocado. Te amamos.

Gracias a todos, Ale

Índice

Capítulo 1: Introducción ... 4
1.1 Objetivos ... 4
1.2 Motivación ... 4
1.3 Desarrollos propuestos ... 6
1.4 Resultados Esperados ... 8
1.5 Estructura del trabajo ... 8
Capítulo 2: Marco Teórico ... 10
2.1 Introducción ... 10
2.2 Diagramas de bloques ... 10
2.3 Programación visual ... 12
2.4 Desventajas de la programación visual ... 19
2.5 Programación de flujos de datos ... 20
2.6 Diagramas de bloques funcionales ... 22
2.7 Microsoft VPL ... 26
2.8 Frameworks ... 27
2.9 Revisión del objetivo ... 33
Capítulo 3: Documentación de Flowgramming Framework ... 36
3.1 Introducción ... 36
3.2 Requerimientos de software ... 37
3.3 Instalación y dependencias del framework ... 37
3.4 Primeros pasos con Flowgramming Framework ... 37
3.4.1 Integración ... 37
3.4.2 Obtener Bloques Disponibles y sus Atributos ... 39
3.4.3 Conectar bloques ... 43
3.4.4 Persistir Diagramas ... 44
3.4.5 Obtener Diagramas persistidos ... 46
3.4.6 Enviar datos a Flowgramming ... 49
3.4.7 Resumen ... 50
3.5 Modos de ejecución en Flowgramming Framework ... 50
3.6 Modos de ejecución y condiciones de los bloques ... 54

3.7 Ciclos ... 57
3.7.1 Tipos de ciclos ... 57
3.7.2 Validación de diagramas e identificación de ciclos ... 61
3.7.3 Seguimiento de errores en ciclos en tiempo de ejecución ... 62
3.8 Configuración del framework ... 62
3.8.1 Instanciación del framework ... 62
3.8.2 Backends de Persistencia ... 64
3.8.3 Backends de Debugging ... 65
3.8.4 Backends de logging de errores ... 66
3.9 Bloques Provistos por el Framework ... 67
3.10 Creación de bloques personalizados ... 72
3.10.1 Introducción ... 72
3.10.2 Estructura del código de un bloque ... 72
3.10.3 Desarrollo de bloques personalizados ... 73
3.10.4 Fields ... 77
3.10.5 Configuraciones opcionales de los bloques ... 82
3.10.6 Bloques para notificación/envío de resultados a sistema general ... 83
3.10.7 Interrupción de ejecución de bloques y diagramas ... 84
3.10.8 Memorización de estado ... 84
3.11 Creación de backends de persistencia ... 85
3.12 Creación de backends de debugging ... 88
3.13 Creación de backends de logging de errores ... 89
Capítulo 4: Notas de implementación de Flowgramming Framework ... 91
4.1 Introducción ... 91
4.2 Variables de Instancia Declarativas ... 91
4.3 Localización de clases ... 92
4.4 Ejecución de diagramas ... 94
4.5 Validación de diagramas ... 95
Capítulo 5: Casos de uso / Casos de ejemplo ... 96
5.1 Introducción ... 96
5.2 Ejemplo de la propuesta #1: Calibración de un sensor de sonido ... 96

5.3 Ejemplo de la propuesta #2: Análisis de Sonido en ciudad eliminando ruidos ... 98
5.4 Ejemplo de la propuesta #3: Detección de fallas en maquinaria según vibraciones. ... 100
5.5 Manejo seguro de una cinta industrial o de un sistema de aerosillas ... 102
5.6 Identificación y conteo de objetos en imágenes ... 103
5.7 Identificación y resaltado de objetos en imágenes ... 105
Capítulo 6: Conclusiones ... 107
Capítulo 7: Trabajos Futuros ... 109
7.1 Introducción ... 109
7.2 Flowgramming como servicio externo ... 109
7.3 Migraciones de modificaciones y eliminación de tipos de bloques y campos ... 110
7.4 Evaluación de ejecución de bloques a través de una tabla ... 110
7.5 Escalabilidad y paralelismo ... 111
7.6 Desarrollo y extensión dirigida por tests automatizados ... 112
7.7 Mejoras de Serialización ... 113
7.8 Señalización de eventos ... 113
7.9 Migración de un backend de persistencia a otro ... 114
7.10 Modos de ejecución a nivel bloque ... 114
7.11 Modo de pruebas aislado ... 114
7.12 Modo de indicar tipos de entrada y salida de un bloque y sus campos ... 115
7.13 Mejoras Menores ... 115
Tabla de figuras ... 117
Referencias bibliográficas ... 120

Capítulo 1: Introducción

1.1 Objetivos

El objetivo del presente trabajo es desarrollar un framework que permita generar modelos de procesamiento de datos en tiempo real a través de diagramas de bloques funcionales [1], una herramienta que provea un conjunto de bloques que al ser relacionados con enlaces direccionales generen diagramas de procesamiento de datos.

El framework propuesto, al recibir un dato correspondiente a una clave especificada, buscará y ejecutará los diagramas relacionados con esta clave, recorriendo y ejecutando cada uno de los bloques de acuerdo al sentido de sus enlaces, produciendo una o varias salidas resultantes del procesamiento de la entrada.

La herramienta debe poder ser fácilmente incluida en sistemas más grandes que se encuentren en desarrollo o ya en producción; es decir, se busca desarrollar una solución que pueda ser utilizada embebida en otros sistemas y no como una aplicación aislada.

El framework proveerá un conjunto inicial de bloques con operaciones y comportamientos simples, y proporcionará sencillos métodos que permitirán al desarrollador generar diferentes tipos de bloques que sean requeridos para las necesidades del sistema en el cual se encuentra embebido.

Como resultado se obtendrá una potente herramienta capaz de resolver diversos problemas en diferentes ámbitos, donde el usuario final (que dispondrá de una variedad de bloques) podrá generar diagramas con los mismos, permitiendo realizar cambios en el procesamiento de los datos de entrada, sin intervención de desarrolladores o programadores especializados. Esto hará posible modificar la lógica en una suerte de programación visual, donde se podrá alterar el comportamiento del software sin ser necesarios los conocimientos requeridos para programar con lenguajes de programación textual, haciendo uso de diagramación de flujos, lo que resulta más natural para cualquier persona ajena al ámbito informático.

1.2 Motivación

En la actualidad existen diferentes herramientas y plataformas que permiten la generación de modelos de bloques funcionales a través de interfaces gráficas, en las que también se pueden generar bloques con comportamiento definido por el usuario.

La mayoría de estas herramientas están elaboradas para ejecutar o diseñar simulaciones de sistemas, que permiten estudiar y evaluar soluciones sin la necesidad de llevar a cabo experimentos costosos. Estas aplicaciones permiten modelar cualquier tipo de sistemas, desde simples aplicaciones de software hasta complejos sistemas mecánicos, electrónicos, eléctricos, etc.

Además de evaluar la construcción de sistemas, pueden también ser utilizadas para evaluar y planificar el desarrollo de cambios en módulos que ya se encuentran funcionando.

La gama de herramientas de simulación a través de bloques funcionales es amplia y la decisión sobre cuál utilizar se debe realizar de acuerdo a las necesidades particulares de cada equipo, recursos y proyecto.

A continuación, se listan algunas de las más conocidas:

- ✓ **Matlab Simulink**: es un entorno gráfico de simulación multidominio a través de diagramas de bloques que permite la generación y uso de librerías con bloques customizados [2] [3]. Se integra con el entorno MATLAB, lo que permite un profundo análisis de los datos obtenidos de las salidas de los modelos generados.
  Provee un conjunto de bloques predefinidos que pueden ser combinados para generar diagramas de bloques, además de la posibilidad de generar bloques customizados. Está dotado de un simulador que permite evaluar y ver el comportamiento del sistema diagramado mientras es ejecutado, complementado con gráficos en tiempo real.
  Es una herramienta de escritorio, para ser usada con MATLAB, con licencia paga y de código privado.

- ✓ **Scilab/XCOS**: al igual que Simulink, es una herramienta provista dentro de un entorno o batería de programas con utilidades para ingeniería y aplicaciones científicas [4]. Se presenta como la alternativa gratuita y libre de Simulink más reconocida. Si bien es una herramienta en desarrollo, por lo que contiene menos librerías o extensiones y la documentación es más escasa, la funcionalidad es similar y la mayor parte de las simulaciones que se pueden realizar con Simulink también pueden ser realizadas con XCOS.
  La licencia de Simulink es bastante costosa, por lo que XCOS representa una muy buena alternativa en muchos casos.

- ✓ **Wolfram System Modeler**: es una alternativa de licencia paga que provee funcionalidades extras no presentes inicialmente en otras herramientas, como por ejemplo visualización mecánica, animaciones en 3D y análisis de ecuaciones [5]. Algunas otras herramientas permiten agregar estas características, pero pagando licencias adicionales por cada una de ellas.
  Además, se caracteriza por no depender de otros entornos, se presenta como una aplicación por si misma (Simulink, por ejemplo, requiere de MATLAB para poder ser ejecutado).

- ✓ **LabVIEW**: es una herramienta focalizada en el hardware. Permite automatizar la validación de diseños, desarrollar equipo industrial y diseñar sistemas de comunicación inalámbrica [6]. Es de licencia paga.

- ✓ **OpenModelica:** es una herramienta de código abierto enfocada en el sector industrial y académico [7]. Está diseñada para el modelado en Modélica [8], un lenguaje de modelado orientado a objetos para el desarrollo de sistemas complejos orientados a componentes [9].

  Una de las principales características que lo diferencia de su competencia es que soporta el diseño de modelos acausales [10], lo que simplificadamente significa que el flujo entre los diferentes bloques es bidireccional y no recorre relaciones direccionadas entre los mismos.

Estas herramientas y muchas otras presentes en el mercado están pensadas para la simulación de sistemas en alto nivel a través de diagramas de bloques funcionales, pero no están pensadas para que estos modelos sean ejecutados directamente, por lo que luego de obtener el diagrama del modelo de simulación, se requiere el desarrollo efectivo/real de la solución para poner en funcionamiento el sistema modelado.

La motivación de este trabajo es realizar un framework que pueda ser fácilmente integrado dentro de un sistema de software más grande, utilizando los diagramas de bloques funcionales para alterar el comportamiento del mismo de forma directa. Esto se llevará a cabo brindando al usuario final la capacidad de modelar diagramas que representan la lógica o el procesamiento que se debe llevar a cabo al recibir determinados datos. Una vez que el diagrama/modelo sea validado y persistido, pasará automáticamente a ser parte de la lógica del software para el cual fue diseñado, procesando los datos de entrada para generar salidas útiles.

En el framework propuesto, el desarrollador (encargado de integrar el mismo) podrá definir nuevos bloques (frente a los requerimientos de los usuarios del sistema final), donde se indique exactamente qué y cómo quiere que los mismos se comporten, ampliando la gama de funcionalidades disponibles según sea necesario; por otra parte el usuario final del sistema podrá utilizar y configurar los mismos para generar nuevos modelos que se ejecuten cuando ciertos datos o señales sean recibidos.

Además, al estar el framework funcionando dentro de un sistema general, los bloques pueden hacer uso directo de cualquier otra parte o módulo del mismo. Por ejemplo, podría hacer uso de una base de datos, llamadas a un ORM que está funcionando actualmente si fuera necesario, utilizar funciones o lógica de la aplicación, o realizar cualquier otro tipo de lógica realizable en un lenguaje de programación convencional, sin ningún tipo de restricción inherente al framework.

1.3 Desarrollos propuestos

Se propone el desarrollo de un framework que permita generar modelos de procesamiento de datos en tiempo real a través de diagramas de bloques funcionales, donde el mismo deberá:

- ✓ **Ser simple**: para el desarrollador deberá ser fácil integrarlo, expandirlo y customizarlo, y que además su utilización resulte en un software que sea fácil de utilizar para el usuario final. Esto implica que tanto el desarrollador como el usuario final no necesiten de una profunda/larga especialización para que la herramienta sea integrada y utilizada.

- ✓ **Ser integrable**: que pueda ser añadido/embebido sencillamente dentro de otros sistemas, sin importar la naturaleza o los fines de los mismos.
  Por ejemplo, debería poder ser embebido tanto en una aplicación de escritorio como en una aplicación web y también en un software aplicado a la industria metalúrgica o ganadera.
  Además, el desarrollador podría definir bloques que hagan uso de funcionalidades externas al framework en sí, propias del sistema que lo embebe.

- ✓ **Proveer programación a través de bloques**: proveer una forma de programar o generar modelos de procesamiento simplemente enlazando o encadenando bloques funcionales, abstrayéndose el usuario final del sistema de implementar código de software.

- ✓ **Ser extensible**: permitir a los desarrolladores crear fácilmente nuevos bloques con nuevas funcionalidades totalmente personalizadas según sea necesario.

- ✓ **Proveer procesamiento de datos en tiempo real**: el framework proveerá los métodos y parámetros de configuración necesaria para recibir entradas o señales que el sistema general o agentes externos envíen, para que los modelos generados ejecuten el procesamiento de los datos a medida que los mismos son recibidos, así como indicar hacia dónde debe dirigirse la salida.

- ✓ **Ser configurable**: el desarrollador podrá hacer uso de diferentes funcionalidades y módulos que el framework provea, como por ejemplo diferentes modos o “backends” para almacenar los modelos y bloques generados. Además, el framework deberá proveer formas sencillas de definir nuevos tipos de “backends”, para que el desarrollador pueda generarlos de acuerdo a sus necesidades. Por ejemplo, para un entorno podría ser útil y suficiente almacenar los modelos en archivos, mientras que, para otro usuario con requerimientos más estrictos de performance, podría ser necesario persistir los modelos en bases de datos en memoria.

- ✓ **Ser “simulable”**: el framework deberá proveer formas de evaluar los modelos con datos de prueba (o con datos reales) y ver los resultados en tiempo real, mostrando los resultados/salida de cada bloque, previo a la persistencia de un modelo que está siendo desarrollado para ser ejecutado con datos reales.

- ✓ **Proveer una sencilla interfaz programable para la producción de entornos visuales para el desarrollo de esquemas de bloques**: el framework deberá brindar una sencilla interfaz de conexión para que el desarrollador pueda diseñar interfaces visuales amigables y fáciles de utilizar para el usuario final del sistema, proveyendo, por ejemplo, la capacidad de generar entornos de simulación Web o aplicaciones de escritorio [11].

El framework permitirá a los desarrolladores integrarlo de la forma que ellos deseen, pudiendo ser aplicado a una infinidad de ámbitos o dominios. Se puede mencionar como ejemplo:

- ✓ Desarrollar un sistema capaz de explotar los diferentes tipos de sensores (acelerómetros, giroscopios, sensores de humedad, de presión, de temperatura, etc.) presentes en dispositivos móviles como celulares, arduinos, raspberries[1], o cualquier otro dispositivo programable que posea sensores, procesando los datos recibidos a través de los bloques anteriormente mencionados.
- ✓ Obtener un software útil para fábricas o entornos de producción en los que se quiere ser notificado si algo está fuera de los límites establecidos (por ejemplo, el ruido en el entorno es muy alto de forma constante, o los niveles de un cierto gas en el aire son muy altos)
- ✓ Desarrollo de un software para el monitoreo de áreas con trastornos o eventos ambientales frecuentes, como tornados, fuertes vientos, terremotos, etc.

1.4 Resultados Esperados

Se espera obtener el framework de modelado de diagrama de bloques funcionales con las características mencionadas en la sección anterior, que pueda ser integrado a cualquier tipo de software, añadiendo una capa de simplicidad que permitirá al desarrollador generar nueva funcionalidad de forma sencilla, y al usuario final diseñar diagramas de una forma natural, sin tener que preocuparse por necesitar conocimientos de programación.

El usuario podrá alterar la funcionalidad o lógica del software solo editando/añadiendo diagramas que permitan alterar los resultados del procesamiento de los datos que el software reciba como entrada.

1.5 Estructura del trabajo

Para finalizar este primer capítulo, se especifica en este apartado la estructura general del presente informe.

[1] Tanto los Arduinos como las Raspebrries, son pequeños dispositivos de hardware programables, microcontroladores con entradas analógicas y digitales, a las que se les puede conectar módulos de hardware de distinto tipo para desarrollar objetos y aplicaciones interactivas.

Se desarrollan a continuación algunos conceptos básicos, el marco teórico y tópicos que han sido investigados y en los cuales se sustenta este trabajo.

Luego se desarrolla la documentación del framework; esto incluye cuáles son sus requerimientos, cómo se instala, recomendaciones y guía de instalación, recomendaciones y guías de integración del mismo, guías para la extensión/personalización, y conceptos y recomendaciones útiles para hacer uso del mismo.

Posteriormente, serán descriptos aspectos del diseño y detalles de implementación del framework, describiendo cómo se desarrollaron las funcionalidades más relevantes y cuáles fueron las técnicas utilizadas.

Para finalizar, se presentan las conclusiones y se mencionan propuestas de trabajos futuros, además un listado con la bibliografía utilizada.

Capítulo 2: Marco Teórico

2.1 Introducción

El objetivo de esta sección es el de brindar al lector un desarrollo básico de cada uno de los conceptos que son necesarios conocer previo a la presentación del framework, ya que algunos de estos son requeridos para entender el entorno en base al cual el desarrollo fue realizado y sobre los cuales se sustenta. Además, posteriormente se justificarán ciertos detalles o decisiones tomadas sobre el diseño del framework, haciendo referencia a varios de los tópicos mencionados en este apartado.

Es necesario que el lector conozca básicamente cómo que es que funcionan los diagramas de bloques, cuáles son sus usos más comunes, la programación visual como alternativa a los lenguajes de programación textual, la programación de flujos de datos y distintas implementaciones, y algunos conceptos formales básicos acerca de qué es, cómo funciona y cómo debe ser diseñado un framework de software.

2.2 Diagramas de bloques

Quizás no sea este uno de los conceptos necesariamente nuevos para el lector, pero es necesario que sea formalizada su definición.

En términos sintéticos, un diagrama de bloques es un tipo de gráfico de flujo de datos utilizado para diseñar nuevos sistemas o para describir o mejorar sistemas ya existentes de forma abstracta [12].

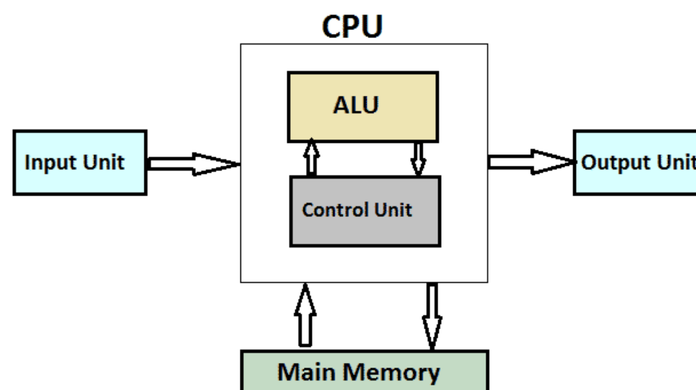

Figura 1: Diagrama de bloques de una CPU

Como se puede observar en la figura 1, se trata de un modelo de alto nivel en el que se muestran los componentes principales de un sistema y las relaciones entre ellos para llevar a cabo su funcionamiento. Su característica de alto nivel permite centrarse en los componentes principales sin entrar en el detalle o la complejidad interna de cada bloque.

En el ejemplo de la figura, se puede observar un diagrama que representa, de forma general, una CPU. Se pueden observar la Unidad Aritmético-Lógica (ALU), la unidad de control y sus conexiones entre sus bloques internos, y sus conexiones con agentes externos como la unidad de entrada, de salida, y la memoria principal.

Debido a su perspectiva de alto nivel, este tipo de diagramas no ofrece un detalle suficiente para implementar la solución, pero se enfoca en las entradas y las salidas de un sistema, permitiendo la comprensión del funcionamiento del mismo a grandes rasgos. Es un tipo de diagrama de "caja negra", por lo que el foco está en ver cómo se relacionan las entradas y salidas de cada bloque, sin importar qué es lo que hace o cómo cada bloque funciona para realizar esta transformación de los datos.

Los diagramas de bloques son útiles en diferentes ámbitos de desarrollo, siendo muy utilizados para la descripción de sistemas de hardware y software. Además, su gama de aplicación es tan amplia que son implementados en un rango extenso de casos que abarca un espectro que incluye esquemas muy simples, diagramas de complejidad media y sistemas de alta complejidad, con un gran número de componentes y relaciones.

Entre los casos de uso más comunes, se pueden encontrar:

- ✓ Presentaciones / modelados: se pueden utilizar los diagramas cuando sólo se busca una abstracción o generalización de diferentes módulos que trabajan en conjunto para resolver un problema. Por ejemplo, en un equipo de desarrollo se podría tener al líder mostrando, a grandes rasgos, diferentes módulos y funcionalidades de un sistema a desarrollar.

- ✓ Software de simulación: para representaciones más formales y menos abstractas, este tipo de diagramas son utilizados en software de simulación de sistemas de software y hardware. Estas herramientas de simulación proveen una formal representación de sistemas, que permiten estudiar y evaluar diferentes soluciones sin la necesidad de llevar a cabo experimentos costosos. Permiten modelar cualquier tipo de sistemas, desde simples aplicaciones de software hasta sistemas mecánicos, electrónicos, eléctricos, etc. La gama de herramientas de simulación a través de bloques funcionales es amplia y la selección se debe realizar de acuerdo a las necesidades particulares de cada equipo, recursos y proyecto. En la sección titulada "Motivación" dentro del apartado de la introducción, se han listado las herramientas con mayor prestigio que caben en esta categoría.

- ✓ Lenguajes de programación visual y programación de controladores lógicos: dentro de esta categoría, se presentan dos tópicos que son de particular interés para este trabajo, por lo que se desarrollarán en los siguientes apartados.

2.3 Programación visual

El objetivo de este trabajo se relaciona directamente con este tópico. Si bien el fin no es obtener o producir un lenguaje de programación visual, el framework desarrollado está pensado para que el usuario final del software pueda modificar la lógica de un sistema de manera gráfica o visual. Por esto, a continuación, se describen algunos aspectos básicos de lenguajes de programación visual, que se basan en un principio muy similar.

Cuando los programadores intentan explicar a alguien qué es lo que un programa hace o debería hacer, generalmente terminan usando un pizarrón o simplemente una hoja para dibujar una representación gráfica del flujo del programa, con cajas y flechas, similar a los diagramas de bloques presentados en el anterior inciso. Esta metodología es aplicada ampliamente debido a su claridad y a su conveniencia, ya que suele ser una herramienta útil para presentar un sistema a cualquier persona, incluso a aquellas que no poseen conocimientos técnicos acerca del dominio.

Entonces, surge una interesante pregunta: ¿por qué no escribir programas de esta forma? Es así es como nacen los lenguajes de programación visual:

> Se denomina lenguaje de programación visual (o VPL, del inglés, visual programming language) a cualquier lenguaje de programación que permita a los usuarios crear programas manipulando elementos gráfica/visualmente, en vez de especificarlos de forma textual, como lo hacen los lenguajes tradicionales. Muchos de los VPL se basan en la idea de "cajas y flechas", donde las cajas u otros objetos de la pantalla son tratados como entidades, conectadas por flechas, líneas o arcos que representan las relaciones existentes entre los bloques o cajas.

La idea de la programación visual resulta muy atractiva, y en algunos campos (como por ejemplo en la enseñanza) resultan muy importantes debido a que es una potente herramienta que enseña a los alumnos cómo desarrollar programas de manera sencilla, haciendo que sus primeros pasos en la programación sean más fáciles que los necesarios para desarrollar software mediante un lenguaje tradicional/textual.

A menudo suele confundirse a los lenguajes de programación visual con lenguajes o entornos que proveen facilidades visuales para la construcción de interfaces gráficas, tales como Visual Studio o las herramientas de desarrollo de Visual C++, Delphi, etc. Pero ellos sólo proveen una facilidad para la implementación de la interfaz visual, mientras que la base de la programación en dichos lenguajes sigue siendo textual.

Los lenguajes de programación visual podrían ser agrupados según sus características de la siguiente manera:

- ✓ **“Scratch” y similares:** el lenguaje “Scratch” y sus derivados son posiblemente en lo que la gente piensa a priori cuando se presentan los VPL. Poseen una gramática que es, en esencia, la gramática de un lenguaje de programación imperativo/textual, pero sus

facilidades gráficas ayudan a combinar de manera más sencilla los elementos básicos como variables, condiciones o los controles de flujo.

Figura 2: entorno de Scratch

Dentro de los bloques, es común encontrar texto que ayuda a describir a los mismos, así como campos textuales que permiten definir nombres de variables y valores. Es posible extender el lenguaje con bloques customizados, que pueden también ser definidos, en una suerte de recursión.

Detrás de la gramática visual, el lenguaje desde sus orígenes ha incluido un entorno de ejecución con renderizado gráfico para crear fácilmente animaciones y juegos.

A comparación de otros VPL, el formato que utiliza es muy similar al de los lenguajes imperativos, excepto por el hecho de usar bloques de una forma que ayuda a descubrir la sintaxis y a prevenir errores, dejando de forma clara y obvia cómo construir instrucciones válidas y cómo los elementos se relacionan entre sí.

La base de este lenguaje puede ser vista como una guía de sintaxis para lenguajes como C o Java.

Entre los lenguajes de programación visual más conocidos que más se asemejan a Scratch, se pueden encontrar: AgentSheets, Ardublock, Blockly, Etoys, graspIO, Hopscotch, Lava, Mama, Microsoft TouchDevelop, Modkit, Scratch, Snap! y Wyliodrin.

✓ **Inspirados en diagramas de flujo:** basándose directamente en la idea de usar "cajas o bloques y flechas" para describir programas, esta categoría de VPLs usa una

representación visual cercana a este tipo de diagramas de flujos para describir el flujo de la ejecución principal.

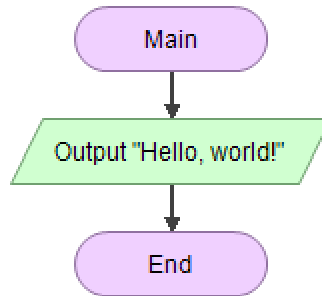

Figura 3:esquema de un VPL inspirado en diagramas de flujo

Representan la secuencia de ejecución entre bloques, con el flujo pasando de un bloque al siguiente, frecuentemente con bifurcaciones que usan el resultado o salida de un bloque para seleccionar qué bloque ejecutar a continuación.

El foco puesto en la ejecución y en una gramática visual simple implica que el lenguaje es fácil de entender, por lo que descubrir la sintaxis no es un problema. Pero las lógicas que este tipo de programas pueden crear son limitadas. Mucho depende de qué es lo que se ejecuta en cada uno de los bloques, que a menudo es provisto y no puede ser alterado desde la interfaz gráfica.

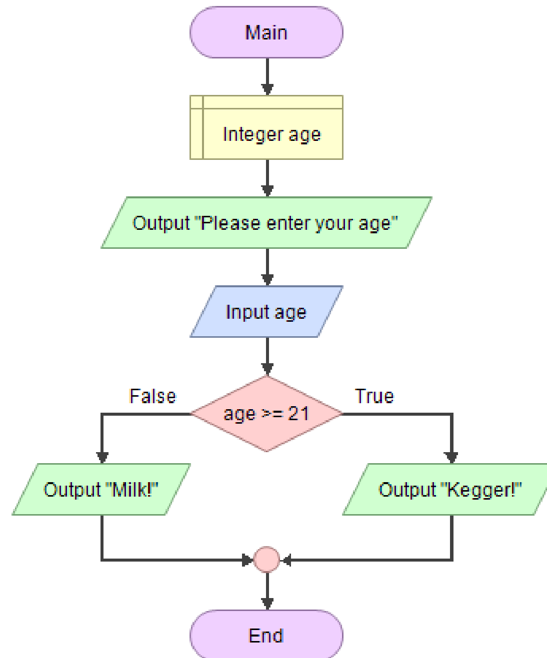

Figura 4: Programa ejemplo de Flowgorithm

Entre los VPL más destacados dentro de este grupo se encuentran: Bonita BPM, Discovery Machine, Flowgorithm, Flowhub, LlamaLab Automate, Node-RED, Raptor, tray.io y WebML.

Algunos formatos similares a los diagramas de flujo proveen una gramática extendida que brindan la habilidad de escribir instrucciones más complejas dentro de la interfaz gráfica. Estos se detallan en los siguientes grupos o categorías.

- ✓ **Programación en base a flujo de datos:** mejor conocido como "dataflow programming", este formato es uno de los más utilizados para aplicaciones profesionales, pensado mayormente para diseñadores de software técnicos, más que para usuarios finales o programadores principiantes.

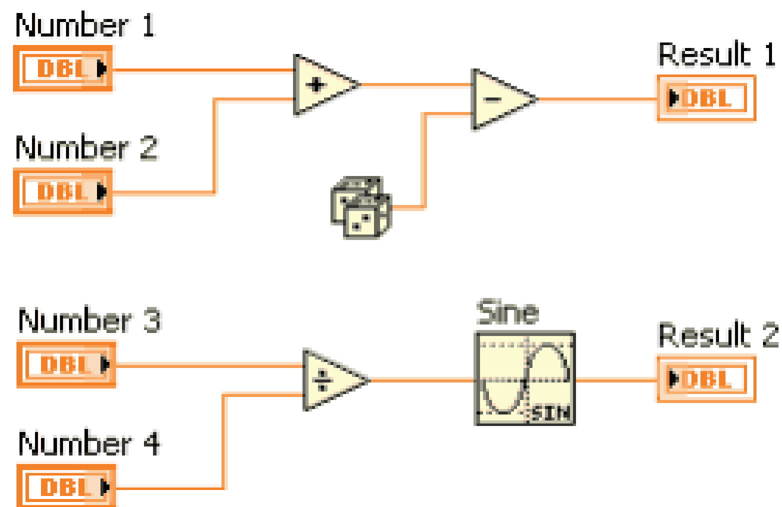

Figura 5: programa de flujo de datos

En este modelo, los bloques representan funciones, y estos son enlazados entre sí a través del flujo de sus datos, a lo largo de sus entradas y salidas. Creando un enlace entre la salida de un bloque y la entrada de otro, se define un flujo de ejecución del programa a través del flujo de los datos.

La gramática visual es bastante simple, se eligen los bloques a usar y se define cómo los mismos se conectan a través de los enlaces. Lo más importante del programa, lo que lo define, se ubica principalmente dentro de los bloques, que son frecuentemente programados con lenguajes textuales comunes o clásicos.

El formato puede ser sencillo, pero requiere conocimientos acerca de la programación de flujos de datos, por lo que está mayormente pensado para usuarios que posean un

proceso de pensamiento técnico, poniendo énfasis en mostrar la estructura principal del programa y el flujo de los datos a través de los bloques, donde el comportamiento de los mismos ya viene dado.

Entre los lenguajes que mejor encajan con esta clasificación, se encuentran: Aldebaran Choregraphe, AudioMutch, CryEngine flow graph, Davinci Resolve, DRAKON, Grasshopper 3D, LabView, NETLab Toolkit, OpenRTM, Quartz Composer, RTMaps, Simulink, Softimage ICE y Spirops AI, Virtools.

Este paradigma es el que más se alinea con la estructura y el objetivo del framework de este trabajo, por lo que será desarrollado con más detalle en la siguiente sección.

- ✓ **Lenguajes de máquinas de estado finito**: permiten definir estados y las transiciones entre estados que son ejecutadas por condiciones. Cuando el estado de un bloque cambia, las instrucciones son ejecutadas. Gráficamente, los bloques representan estados y los enlaces representan las transiciones.

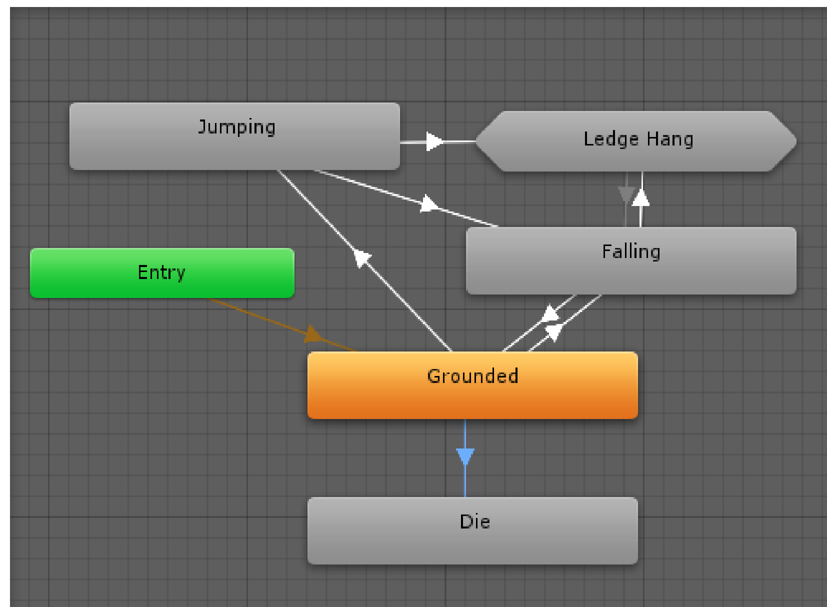

Figura 6: Programa realizado con un VPL de máquina de estado finito

El usuario diseña el flujo de ejecución de forma similar a los diagramas de flujo, enlazando estados y definiendo las condiciones que ejecutan el cambio de estado. La diferencia radica en que la lógica se controla por las transiciones de los estados y sus condiciones y no por lo que se encuentra dentro de los bloques, por lo que provee la habilidad de ejecutar o crear instrucciones más complejas.

Sin embargo, este tipo de lenguajes requiere que el usuario entienda y manipule las condiciones de cambio de estado con expresiones textuales, por lo que es un poco más complejo de entender y posee más texto que los diagramas de flujo, aunque a la vez esto los hace un poco más expresivos. Requieren de un conocimiento técnico más profundo que los otros tipos de VPL.

En esta categoría se encuentran lenguajes como EKI One, NodeCanvas, Unity3D Mecanim Animator Controller, y xaitControl.

- ✓ **Árboles de comportamiento:** Similares a la idea de las máquinas de estado finito y otros de los paradigmas, la gramática visual de estos lenguajes se compone por bloques y enlaces que define el flujo de ejecución. Sin embargo, los lenguajes ubicados bajo esta categoría, evalúan los bloques en forma descendente, y retornan un estado al bloque previo a través de su estructura de árbol.

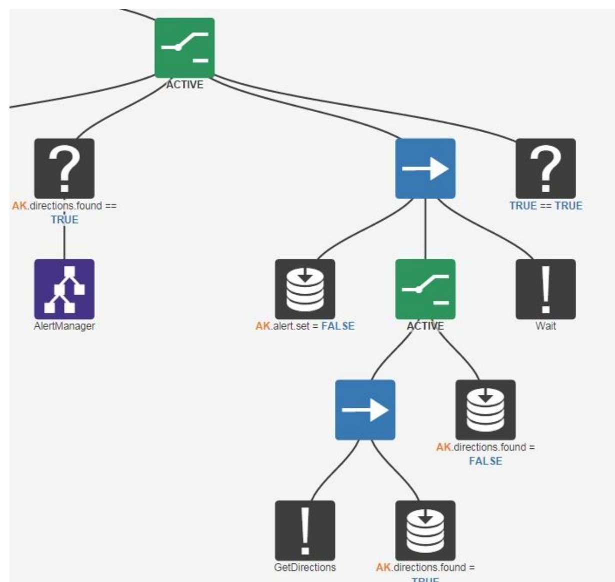

Figura 7: árbol de comportamiento

Cada bloque define comportamiento que depende de sus bloques "hijos" en el árbol. Por ejemplo, un bloque de "secuencia" evalúa sus descendientes en orden y se detiene si alguno de ellos falla.

La edición de programas en esta modalidad es más directa que en las máquinas de estado finito y se puede realizar lógica más compleja creando enlaces entre bloques. También es más legible debido a que poca lógica depende de los valores de los campos textuales.

La gramática visual es sencilla y fácil de descubrir, aunque el usuario necesita entender la manera en que los árboles son ejecutados, que es similar a entender el concepto de "pila de ejecución" en la programación clásica.
Algunos ejemplos de lenguajes que se encuentran en esta categoría son: {Rival Theory} RAIN, AngryAnt Behave, Behavior3, NodeCanvas, y craft ai.

- ✓ **Reglas basadas en eventos**: es un tipo de VPL un poco más simple: los usuarios definen reglas clásicas de lógica del tipo "si ocurre 'x' evento, entonces realizar 'y' acción". Una regla es ejecutada cuando la condición se cumple, ejecutando instrucciones específicas. Los elementos visuales son frecuentemente básicos, y la mayor parte de los bloques pueden ser utilizados tanto para las condiciones como para las instrucciones.

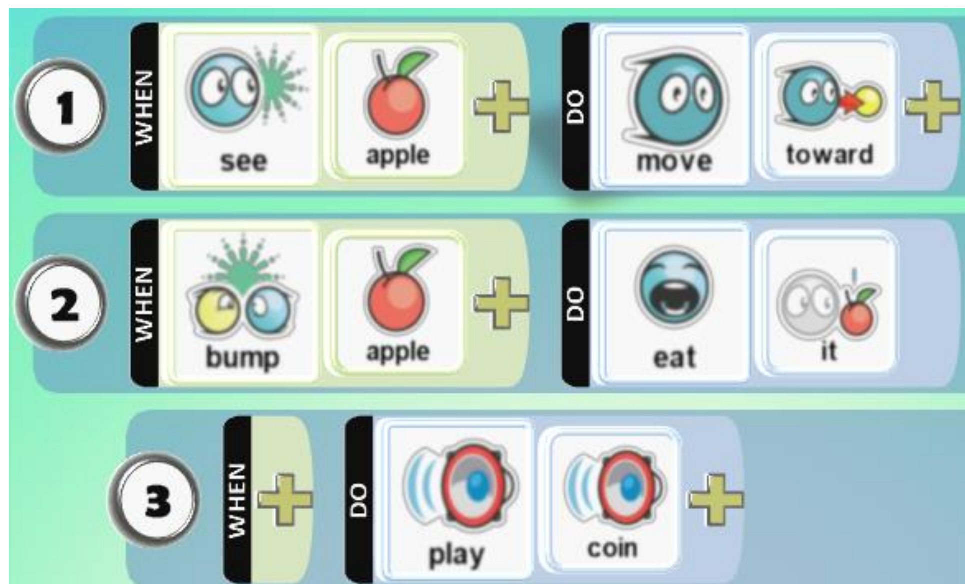

Figura 8: reglas basadas en eventos

La ventaja y a la vez desventaja de este formato es la simplicidad. El lenguaje es tan reducido que se podría argumentar que no es realmente un lenguaje de programación, pero por otro lado esto es lo que lo hace fácil de usar para usuarios que nunca han desarrollado un programa.

Generalmente los usuarios pueden personalizar sus aplicaciones con reglas hechas a medida. Los programas que pueden ser escritos con esta modalidad son limitados, pero están pensado para que los "usuarios finales", poco especializados, puedan tomar el control.

Bajo esta categoría, se puede hallar Construct 2, IFTTT, Kodu (pictured above), Netvibes Dashboard Of Things y Zapier.

Aunque el uso de los VPL no está generalizado, existen ciertas áreas en las que han tomado relevancia, a punto tal, que se puede decir que son suficientemente maduras en cuanto a su uso. En estas áreas, tales como la programación de PLCs (que se desarrollará luego), los VPL se presentan como herramientas eficientes, debido a que sus ventajas sobre la programación clásica los hacen realmente útiles.

Además, en áreas fuera de la producción de bienes y servicios, como por ejemplo en la educación, los profesores utilizan los lenguajes visuales como una forma accesible o simplificada para enseñar aspectos básicos de programación. Como ya se ha mencionado, el lenguaje Scratch es útil para este fin, y se ha adoptado de forma generalizada.

En algunas áreas relacionadas a la producción multimedia, los diseñadores los usan para sus labores profesionales, especialmente aquellos lenguajes que les permiten estructurar los programas a través de la utilización de bloques que pueden ser reusados, de una forma que provee legibilidad, ayudando a simplificar la complejidad.

Los VPL también son utilizados en otros ámbitos donde su uso está pensado para usuarios finales, particularmente para que puedan personalizar sus aplicaciones o crear nuevas reglas. El formato en general es basado en eventos para facilitar su uso, donde el usuario indica qué hacer ante la ocurrencia de un evento particular.

Para aplicaciones pensadas para "creadores" o "invención", también los VPL son utilizados, proveyendo una forma más directa de programación. Por ejemplo, se utilizan para programación de robótica o creación de juegos.

Los VPL han tomado territorio poco a poco, y son cada vez tenidos más en cuenta por su accesibilidad y porque ayudan a controlar la complejidad de los sistemas de una forma que permite un sencillo mantenimiento.

2.4 Desventajas de la programación visual

Incluso cuando los VPL ofrecen una forma más sencilla de aprender y a pesar de que reducen las dificultades de la sintaxis en comparación a los lenguajes tradicionales, todo tipo de lenguaje requiere que el usuario se familiarice con los conceptos básicos de la programación, como por ejemplo el concepto de variable y las generalidades de la programación imperativa. Por lo que se necesita que el usuario sea dotado de algunos conceptos básicos de programación. Esta desventaja es relativamente leve, ya que si el lenguaje visual es sencillo, dotar al usuario de estos conceptos básicos no debería llevar demasiado tiempo, e incluso podría ser intuitivo si el entorno tiene en cuenta este aspecto.

Pero si por un momento se quita el foco de los usuarios no especializados y se llevan los VPL al ámbito de los programadores profesionales que escriben programas día a día, la programación visual parecería no agregar demasiado valor. Cuando un desarrollador está familiarizado con la sintaxis de un lenguaje y necesita trabajar con diferentes tipos de componentes o librerías, las ventajas de los VPL no resultan muy interesantes.

Además, en términos de representación de expresiones, el texto es compacto y abierto: muchas palabras pueden situarse en lugar de lo que el espacio de un bloque de un VLP ocuparía, y además se puede referir sencillamente a funciones externas y extender el lenguaje añadiendo palabras clave, mientras que en los VPL esta funcionalidad es más limitada y a veces no está explicita. Incluso para los VPL que combinan elementos visuales con texto, su representación visual no puede competir con la densidad de información que el texto provee. Por lo tanto, incluso teniendo en cuenta las ventajas de los VPL, para los desarrolladores profesionales no resultan ser una alternativa mejor a la de programación clásica o textual.

Luego, en la mayor parte de los casos, los entornos de los VPL son especializados, diseñados para un dominio en particular como por ejemplo el diseño de juegos, dificultando su uso en áreas diferentes.

Otro factor importante de los VPL es que poseen algunas limitaciones de rendimiento, en particular la velocidad de ejecución y la escalabilidad suelen ser el mayor problema.

Además se puede hacer mención de otros aspectos en los que la programación textual es más fuerte, como por ejemplo en la documentación, el nombrado de variables y objetos para distinguir elementos que son del mismo tipo, y la expresión de conceptos que son inherentemente textuales, tales como fórmulas algebraicas.

2.5 Programación de flujos de datos

En el ámbito informático, el concepto de “flujos de datos” o “dataflow” puede ser considerado de diferentes formas:

- ✓ **Como arquitectura de software:** se lo utiliza para hacer referencia a una arquitectura de software basada en la idea de separar una cadena de actores del procesamiento computacional en diferentes etapas o tareas que pueden ser ejecutadas de forma concurrente.

- ✓ **Como arquitectura de hardware**: nace como una alternativa en contraste a la arquitectura tradicional de Von Neumann, donde muchos conceptos cambian, siendo el cambio más relevante que la ejecución de instrucciones no es secuencial ni en base a un contador de programa, si no que se basa en que las instrucciones son ejecutadas tan pronto como los argumentos de las mismas estén disponibles, por lo que el orden de

ejecución de las instrucciones es impredecible o no determinístico. Este tipo de hardware se utiliza en campos especializados de procesamiento y no en computadoras de uso general.

- ✓ **Concurrencia:** en el ámbito del procesamiento o programación concurrente, se denomina red de flujo de datos a una red que ejecuta procesos de manera concurrente, todos ellos comunicando datos a través de canales, mediante la técnica de pasaje de mensajes.

Para el interés de este trabajo, los flujos de datos como arquitectura de software es el significado más relevante. De hecho, la programación de flujos de datos o "dataflow programming", se trata de un paradigma que modela los programas de forma gráfica, donde se detalla cómo los datos fluyen a través de diferentes bloques o funciones que los procesan, implementando los principios de arquitectura de los diagramas de flujo de datos. Por lo tanto, el énfasis está puesto en ver como los datos fluyen entre las diferentes funcionalidades que los procesan, modelando el esquema a través de una serie de conexiones.

Desde los inicios de la programación, un programa clásico o tradicional fue y es modelado como una serie de operaciones que se ejecutan en un orden específico. En el ámbito de la informática, se refiere a este modelo a través de conceptos como programación secuencial, procedural, imperativa, o de control de flujo.

En contraste con la programación clásica, un programa de flujo de datos pone el énfasis en el movimiento o seguimiento de los datos a través de una serie de enlaces que conectan funciones o bloques de "caja negra", donde cada bloque es ejecutado tan pronto como todas sus entradas son válidas.

El conjunto de operaciones o bloques conectados en serie, donde la salida de un bloque es la entrada del siguiente, se denomina pipe, y provee una diferencia en cuanto al concepto de estado en comparación con la programación tradicional.

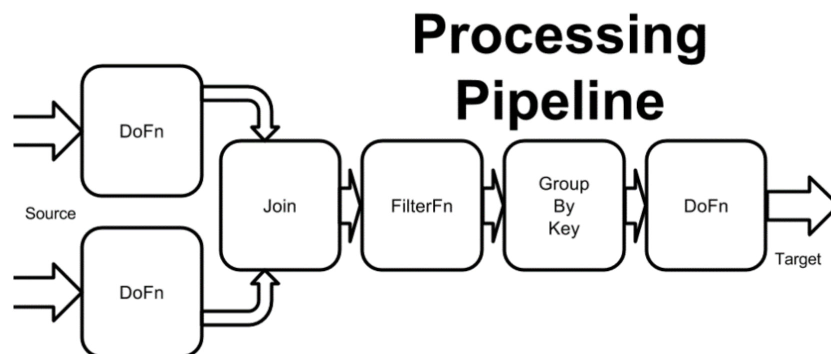

Figura 9: pipe de procesamiento de flujo de datos

En la programación secuencial, cada programa puede ser visualizado como un único “hilo” o “trabajador" que va ejecutando diversas tareas y donde este tiene que guardar estado acerca de las diferentes tareas que está ejecutando en un determinado momento.

En cambio, en el paradigma de la programación de flujo de datos, un pipe puede ser visto como una serie de "trabajadores" en una línea de ensamblado, donde cada uno ejecuta una tarea cualquiera. Cada una de estas tareas necesita solo los datos de entrada para ser ejecutada, por lo que siempre que un trabajador no esté ocupado ejecutando una tarea, está listo para ejecutar una nueva, dejando de ser relevante el concepto de estado “general” del programa.

Esto resulta de gran interés cuando existen operaciones o procesamiento de datos extensos, donde cada tarea puede ser ejecutada en un "trabajador" particular, permitiendo el procesamiento paralelo y distribuido de datos en diferentes "trabajadores". Para que el paralelismo pueda ser ejecutado, solo hace falta compartir una estructura entre los procesos “trabajadores”, que permita a cada uno identificar una tarea que esté lista para ser realizada (es decir, una tarea en la cual todos sus datos de entrada ya se encuentren disponibles para ser procesados).

2.6 Diagramas de bloques funcionales

Ya teniendo en cuenta los conceptos de diagramas de bloques y programación visual, sus ventajas y desventajas, y la programación de flujos de datos, y antes de comenzar a desarrollar los conceptos y detalles del framework que este trabajo provee, se desarrolla en esta sección una interesante aplicación de programación visual a través de bloques.

El diagrama de bloques funcionales o más conocido como FBD (del inglés Function Block Diagram) es un lenguaje de programación gráfico basado en diagramas de bloques, originalmente pensado para Controladores Lógicos Programables[2] (o PLCs), que surge como alternativa al lenguaje “ladder” (el original lenguaje de estos controladores), buscando brindar programas más claros y legibles, sobre todo para la representación de diagramas con funcionalidades complejas, evitando además la necesidad de crear variables y registros adicionales o intermedios.

---

[2] Un controlador lógico programable, más conocido por sus siglas en inglés PLC (Programmable Logic Controller) o por autómata programable, es una computadora utilizada en la ingeniería automática o automatización industrial, para automatizar procesos electromecánicos, tales como el control de la maquinaria de la fábrica en líneas de montaje o atracciones mecánicas.

Los PLC son utilizados en muchas industrias y máquinas. A diferencia de las computadoras de propósito general, el PLC está diseñado para múltiples señales de entrada y de salida, rangos de temperatura ampliados, inmunidad al ruido eléctrico y resistencia a la vibración y al impacto. Los programas para el control de funcionamiento de la máquina se suelen almacenar en baterías, copia de seguridad o en memorias no volátiles.

Se desarrolla a continuación algunos detalles básicos de este lenguaje, ya que el tipo de lógica del mismo resulta interesante y es una de las bases en torno a las cuales el framework de este trabajo ha sido desarrollado.

Se trata de un lenguaje que describe detalladamente las funciones existentes entre variables de entrada y de salida dentro de un sistema. Cada una de estas funciones están compuestas por un conjunto de bloques, donde las variables y los bloques son unidos mediante líneas o enlaces, conectando dos puntos lógicos dentro del diagrama, donde los siguientes enlaces son posibles:

- ✓ Una variable de entrada con la entrada de un bloque
- ✓ La salida de un bloque con la entrada de otro bloque
- ✓ La salida de un bloque con una variable de salida

Los enlaces que conectan a los bloques son orientados o direccionados, lo que significa que indican la dirección en la que los datos fluyen, y donde cada extremo del enlace debe representar datos del mismo tipo.

Cada bloque funcional representa una unidad de instrucción, que al ser ejecutada produce una o más salidas a partir de sus entradas. La forma básica de representar un bloque es de la siguiente manera:

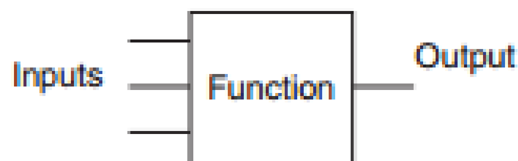

Figura 10: estructura de un bloque de diagrama de bloques funcionales

Un bloque funcional se representa como una caja rectangular con sus entradas a la izquierda y sus salidas a la derecha y dentro del bloque se representa, textualmente, la función que este ejecuta.

Una de las ventajas principales de este lenguaje es que tiende a ser fácil de seguir, ya que simplemente hay que seguir los caminos que existen entre los diferentes bloques y variables.

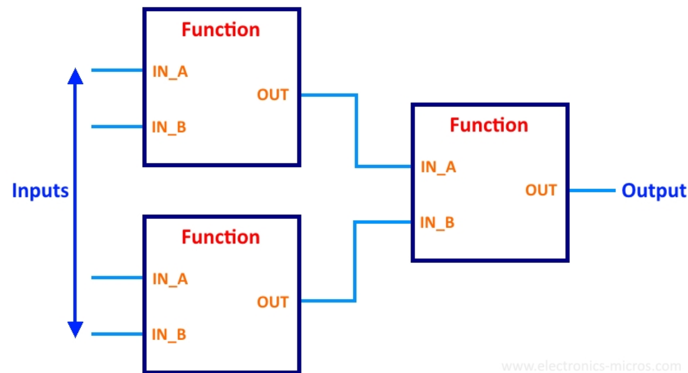

Figura 11: Diagrama de bloques funcionales

Los bloques de un programa de un PLC pueden ser clasificados en muchos tipos, hay bloques de control, de almacenamiento, y otros más, pero los bloques funcionales (en los que está el interés de este trabajo), pueden ser sub-clasificados en dos grupos generales:

- Bloques de función: son bloques con una memoria interna que el usuario puede definir
- Funciones: son rutinas de uso frecuente que se encuentra disponible para todos los usuarios y programas (como por ejemplo operaciones lógicas o aritméticas)

Sin embargo, esta nomenclatura o clasificación puede variar dependiendo del fabricante del PLC y las herramientas que este provea.

La ejecución de los programas en los PLCs es cíclica; una vez que el programa se encuentra descargado en un PLC y este último es encendido, se evalúa el programa completo hasta que termine, y luego se comienza la ejecución una y otra vez.

En realidad, con un mayor nivel de detalle, este ciclo se descompone en algunos pasos adicionales: cuando un PLC es encendido, primero realiza algunos chequeos de hardware y software en búsqueda de fallas. En caso que no haya ninguna falla, comienza su ciclo se de escaneo, que consiste de tres fases:

1. **Escaneo de entrada:** se realiza un “snapshot” o captura de las entradas, verificando para cada tarjeta o módulo de entrada sus valores y se los almacena en una tabla para ser usados en el próximo paso. Esto hace que el proceso de ejecución sea más rápido y evita cambios en las entradas mientras el programa es ejecutado.
2. **Ejecución del programa:** el PLC ejecuta el programa de a una instrucción a la vez, usando la copia de los valores de entrada almacenados en memoria en el paso anterior.
3. **Escaneo de salida:** una vez finalizada la lógica del programa, las salidas son almacenadas de forma temporal en memoria. El PLC actualiza el estado de las salidas en memoria con los resultados del programa.

Una vez finalizado, empieza el ciclo nuevamente, desde la comprobación de errores hasta el almacenamiento de los resultados de la ejecución.

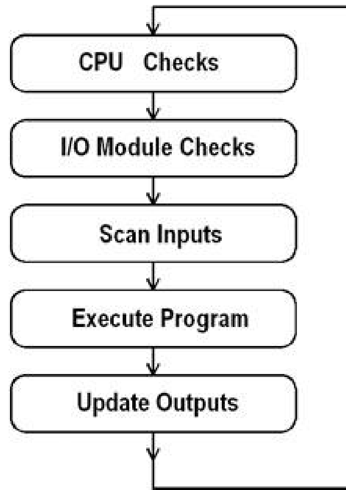

Figura 12: Ciclo de ejecución de un programa en un PLC

Este tipo de ejecución tiene como consecuencia que eventos que ocurren en un período muy corto de tiempo puedan ser no tenidos en cuenta o despreciados. Por ejemplo, si el valor de una entrada cambia su estado por un período de tiempo muy corto mientras el programa es ejecutado, este cambio en el valor podría no ser tenido en cuenta.

Los valores de las diferentes variables pueden ser utilizados entre diferentes programas o para mostrar gráficos o alertas en una interfaz. Dependiendo de la marca/fabricante de cada PLC, existen diferentes nomenclaturas y jerarquías de aplicaciones ejecutables, encontrando redes, tareas, rutinas y programas.

A continuación, se muestra un simple ejemplo de un programa de un PLC desarrollado con diagramas de bloques funcionales, que ejecuta la siguiente función:

MW4 = ((IW0 + DBW3) x 15) / MW0

Donde MW4, IW0, DBW3 y MW0 son variables o “tags” del PLC.

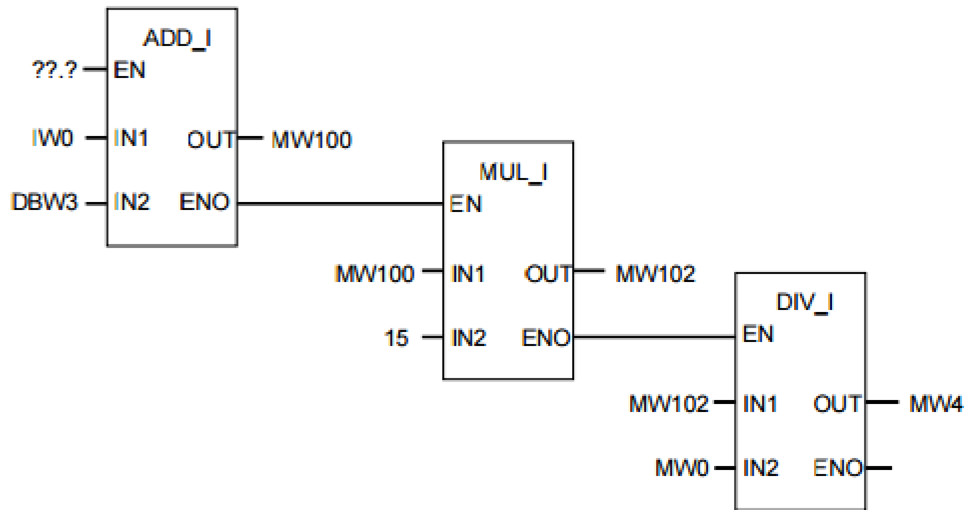

Figura 13: Programa de ejemplo de un PLC

La variable IW0 se suma con DBW3, y la suma es almacenada en la variable MW100. Luego el dato almacenado en MW100 es multiplicado por 15, almacenando el valor en MW102, que luego es dividido por el valor de MW0, almacenando el resultado final del programa en MW4.

2.7 Microsoft VPL

Este apartado, brevemente, describe un lenguaje que utiliza conceptos similares a los de la programación de bloques funcionales, sin estar necesariamente pensado para procesamiento de grandes cantidades de datos.

Microsoft VPL, es un lenguaje de programación visual producido por Microsoft, para su entorno de programación de robótica. Si bien se podría decir que es similar a muchos otros VPL, se desarrolla un poco más en profundidad debido a las similitudes en cuanto a funcionamiento que tendrá con el framework propuesto.

Se trata de un lenguaje de programación pensado para programadores principiantes, donde el entorno de desarrollo se inspira en la programación de flujos de datos. Siguiendo este modelo, se basa en evitar el uso de comandos que se ejecutan de forma secuencial, implementando una serie de "trabajadores" que ejecutan tareas tan pronto como las entradas que las mismas necesitan están disponibles.

Un flujo de datos programado en este lenguaje consiste en una serie de actividades o tareas que son representadas como bloques con entradas y salidas que son conectadas con otras tareas:

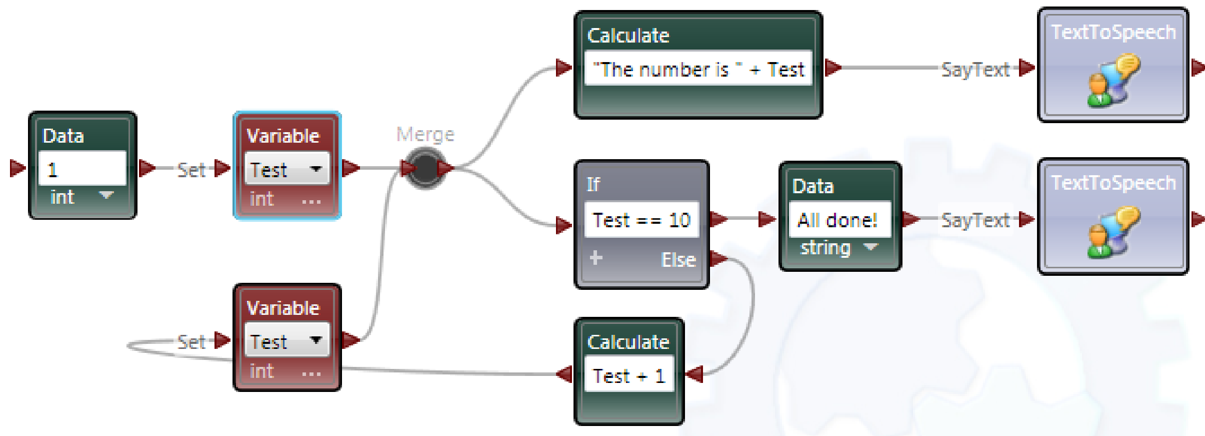

Figura 14: Programa ejemplo de Microsoft VPL

Existen diferentes tipos de actividades o tareas, pudiendo estas representar funciones de procesamiento o control de flujos, y también se pueden desarrollar actividades con comportamiento definido por el usuario. Las conexiones envían los datos desde y hacia las tareas utilizando técnicas de pasaje de mensajes.

Un bloque incluye un nombre que indica su tipo y "pines" de conexión. Los pines de conexión son los que permiten enlazar distintos bloques. Un pin a la izquierda representa un pin de entrada, mientras que uno a la derecha es de salida. Un bloque puede poseer varios pines de entrada, cada uno con su propio set de pines de salida.

Los pines de salida pueden ser de dos tipos: de respuesta o de notificación. Los de repuesta envían datos que son el resultado del procesamiento de la entrada, mientras que los de notificación envían información acerca de cambios en el estado interno del bloque.

Este lenguaje y la forma en que el mismo funciona es similar al framework de este trabajo, ya que ambos utilizan el paradigma de dataflow, pero se diferencian en cuanto a que el objetivo en el caso de este trabajo, es el de brindar la posibilidad de realizar programas que no necesariamente van a realizar el procesamiento de grandes conjuntos de datos en diferentes tareas de forma concurrente o paralela, aunque esto se propondrá como un trabajo a futuro.

2.8 Frameworks

Este apartado está dedicado al desarrollo de algunos conceptos acerca de los frameworks. Si bien muchos desarrolladores frecuentemente hacen uso de ellos, e incluso de forma implícita conocen muchas de las técnicas utilizadas, no siempre se detienen a pensar exactamente qué es lo que pasa por detrás, o como es la lógica mediante la cual un framework es desarrollado, y, por lo tanto mediante la cual funciona.

Si se busca la definición de framework en algún diccionario, se puede encontrar:

> "Una estructura que soporta o encapsula algo más, especialmente un soporte esquelético usado como la base para algo que se quiere construir" [27]

En el ámbito informático un framework es básicamente eso. Es decir, puede ser definido como un esqueleto pre-desarrollado de una aplicación de software, diseñado para ser extendido y configurado por desarrolladores, en la búsqueda por satisfacer un requerimiento/objetivo o crear una nueva aplicación.

El propósito de los frameworks es el de mejorar la eficiencia a la hora de producir un software nuevo. Pueden mejorar la productividad y la calidad, confiabilidad y robustez de un nuevo software. Particularmente, la productividad es mejorada al permitir que los desarrolladores pongan el foco en los requerimientos particulares de su aplicación, en vez de consumir tiempo en el desarrollo de la infraestructura de la aplicación, o en lógicas que se repiten de software en software, evitando al desarrollador "reinventar la rueda". Por ejemplo, para un proyecto web de publicaciones de artículos periodísticos, los desarrolladores pueden centrarse en cómo será la carga de estos artículos, delegando al framework el manejo de los pedidos ("requests") y el manejo del estado del sitio.

Si bien es común confundir a los frameworks con un conjunto de librerías, en realidad son conceptos diferentes. Una librería, en esencia, provee un conjunto de objetos y funciones que el desarrollador debe conocer y saber cómo utilizar para alcanzar los objetivos buscados. En un framework, en cambio, el desarrollador es el encargado de crear estos objetos y funciones que son propios del dominio de la aplicación, y es el framework quien los llama o utiliza, es decir que el framework es quien define el flujo de control de la aplicación. Esto es lo que se conoce como inversión de control (IoC).

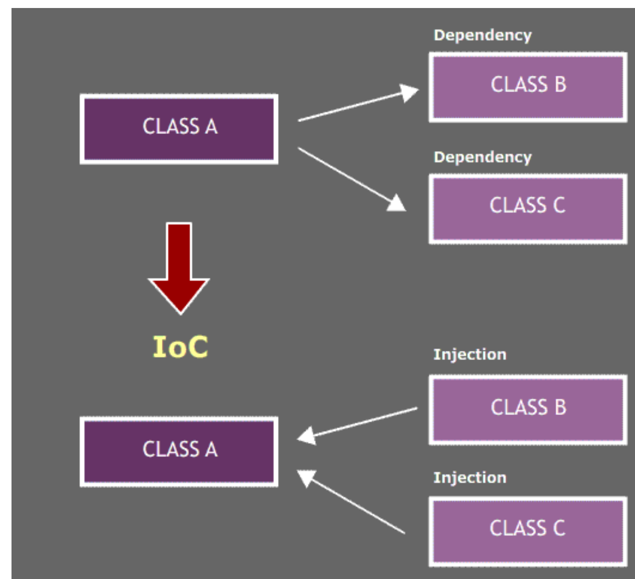

Figura 15: Inversión de control

Generalmente, los frameworks proveen una forma de sobrescribir o extender las características o funcionalidades provistas por el mismo. Una forma común de realizar esto, es aquella en la que el framework define métodos abstractos o virtuales que el desarrollador debe implementar, de la misma forma que se pueden crear objetos que poseen una interfaz definida, y que el framework sabe cómo controlar. Estas técnicas permiten sacar ventaja del polimorfismo, para interactuar con software desarrollado por otras personas. Además, se pueden definir varias formas de llevar a cabo una tarea.

Se muestra a continuación un ejemplo simple pero concreto del uso de un framework real: Flask[29]. Flask es un micro-framework web para Python, diseñado para realizar aplicaciones web “livianas”.

Para retornar una simple página con Flask, y que esta corra en un servidor local, es suficiente con escribir:

```
from flask import Flask
app = Flask(__name__)


@app.route('/')
def hello_world():
    return 'Hello, World!'
```

Figura 16: Código ejemplo de Flask #1

En este ejemplo, se instancia una aplicación Flask, y se define un controlador que devolverá una página con el contenido “Hello World”. En este caso, el potencial del framework no es tan fructífero o no provee demasiado, puesto que se podría escribir una simple página HTML para cumplir el mismo objetivo. Ahora bien, si que se quiere desarrollar un controlador para el login de un usuario a la aplicación. Esto puede resolverse de la siguiente manera:

```
from flask import request
from flask import render_template


@app.route('/login', methods=['POST', 'GET'])
def login():
    error = None
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        if valid_login(username, password):
            return log_the_user_in(username)
        else:
            error = 'Invalid username/password'

    return render_template('login.html', error=error)
```

Figura 17: Código ejemplo de Flask #2

En este último ejemplo, se puede observar que el framework se encarga de muchas más cosas, sin entrar en detalles ni ver qué hace cada método en particular, la idea es simplemente mostrar el potencial que provee un framework. A grandes rasgos se puede ver que el controlador se define para el servidor en la URL "/login", y que permite métodos HTTP POST y GET. Además, brinda un objeto "request" del cual el desarrollador puede obtener parámetros que el usuario ingresó en un formulario web. Para concluir, existe un renderizado de una página a través de una facilidad que el framework provee.

Si se compara este último ejemplo con el esfuerzo y la cantidad de líneas de código requeridas para desarrollar todo el manejo de requests, el contenido y los parámetros recibidos por estas, el método mediante el cual se aceptan solo ciertos métodos HTTP, la lógica necesaria para renderizar una página HTML y mostrar mensajes enviados desde el controlador, el enrutamiento de un controlador con una URL, etc., se puede notar a simple vista la gran ventaja y todo el potencial que brinda el framework frente al desarrollo desde la base sin hacer uso de este, incluso en el caso en que se hiciera uso de librerías.

Sintetizando, se puede decir que hay al menos tres elementos claves que distinguen o separan a un framework de una librería:

- **Inversión de control:** en un framework, de forma opuesta a las librerías, el control del flujo principal no está manejado por quien hace uso del framework, si no por este último.
- **Extensibilidad**: el framework provee los medios para que el desarrollador pueda definir su código especializado, del cual el framework hará uso.
- **Código no modificable:** el código "core", es decir, la base que lo hace funcionar, generalmente no debe ser modificado. Si bien algunos frameworks permiten modificar

algunas secciones “críticas”, aquellas que llevan a cabo la lógica más importante deben conservarse intactas. Además, la edición de secciones que no deben ser alteradas, podría traer problemas a futuro si se quiere actualizar la versión del framework.

Si bien existen frameworks que son pequeños o simples, la mayor parte son muy completos y complejos, lo que hace que el tiempo de aprendizaje pueda ser un poco extenso. Es por eso que a la hora de utilizar uno, se debe pensar en hacerlo de forma subsecuente y continua. Caso contrario, si se lo utiliza para algo muy puntual, puede que el tiempo invertido en el aprendizaje del framework cueste más que si el equipo desarrolla un código particular para ese propósito específico.

De todas formas, una vez que un framework es aprendido, los proyectos pueden ser mucho más fáciles y rápidos de concluir y en algunos casos no es necesario conocerlo de forma completa, sino que con algunos detalles puede ser suficiente. Como ya se ha expresado, la idea de los frameworks es la de brindar un conjunto de soluciones que una vez conocidas y con las cuales el desarrollador se encuentra familiarizado, permiten incrementar la eficiencia en la producción de software de forma automática.

Además, los frameworks pueden ser de uso general o específicos para un dominio. Por ejemplo, un entorno como Visual Studio es un entorno de uso general, ya que provee las herramientas para generar (entre otras cosas) aplicaciones de escritorio. Es decir, el usuario se enfoca simplemente en definir el diseño de las pantallas y sus elementos, y qué hacer cuando el usuario interactúa con ellos, o ante la ocurrencia de ciertos eventos. Este tipo de frameworks son de alta utilidad para el desarrollador, ya que, de otra forma, este tendría que encargarse de la tediosa y larga tarea de crear las ventanas, diseñar los objetos, etc.

Sin embargo, estos frameworks se definen como de uso general, debido a que con ellos se puede crear (en este ejemplo), cualquier tipo de aplicación de escritorio. En cambio, existe otro tipo, enfocado en un dominio en particular, que permite al desarrollador centrarse en detalles más específicos. Por ejemplo, suponiendo que se quiera desarrollar un sistema de ventas de productos, se podría utilizar un framework en el que el desarrollador se enfoque en los tipos de productos y los campos o atributos de estos, y no por toda la lógica de compra, registración, stock, etc.

Sintetizando lo expresado, entre las ventajas de usar un framework se puede encontrar:

- ✓ Reusabilidad de código que ya ha sido pre-desarrollado y pre-testado. Esto permite incrementar la confiabilidad de la nueva aplicación y reducir tiempo y esfuerzo de testeo.
- ✓ Relacionado con lo anterior, también surge la ventaja de poseer muchas características o requerimientos críticos pre-desarrollados, requerimientos en los que el desarrollador podría cometer errores. Por ejemplo, en un framework web estas características son, entre otras, las de seguridad.

- ✓ Un framework puede ayudar a establecer mejores prácticas de desarrollo y usos de patrones de diseño y de nuevas herramientas de programación.
- ✓ Una actualización de versión de un framework puede proveer nueva funcionalidad, mejorar la eficiencia o mejorar la calidad sin intervención o desarrollo adicional.
- ✓ Por definición, un framework provee los medios para extender su comportamiento.

Entre las desventajas de su uso se encuentran:

- La creación de un framework es difícil, requiere de experiencia en el ámbito o dominio para el cual se lo construye, y consume mucho tiempo (lo que se traduce en dinero)
- En algunos casos, la curva de aprendizaje de un framework puede ser bastante pronunciada
- Con el tiempo y las mejoras, los frameworks pueden volverse muy complejos
- Para casos muy específicos donde requerimientos de eficiencia críticos o lógica de aplicación muy compleja, pueden resultar de escasa utilidad o costosos en términos del tiempo invertido para su aprendizaje.

Por esto, a la hora de tomar la decisión de usar o no un framework, o de seleccionar uno entre una gama de opciones, es altamente recomendable acudir a un experto y/o evaluar todas las características y funcionalidades de los mismos y compararlos con los requerimientos del software que se buscar implementar.

Arquitectura de un framework

Como ya se ha mencionado, el framework es el encargado de definir funcionalidades que deben y otras que no deben ser extendidas. Esto, en términos arquitectónicos del diseño de un framework, es lo que se conoce como "frozen spots" y "hot spots" (puntos o secciones congeladas y puntos calientes).

Los frozen spots, definen la arquitectura estructural del framework, es decir, define los componentes básicos y las relaciones entre ellos. Estos puntos son lo que se deben mantener sin cambios, o "congelados" cuando el framework es instanciado.

Los hot spots, en cambio, representan aquellas secciones en las cuales los programadores incluyen su código propio para definir la funcionalidad específica que un proyecto en particular requiere.

También, a nivel conceptual, existe un término denominado "hook". Los "hooks methods", son aquellos métodos definidos por el desarrollador, de los cuales el framework hace uso, y además

define la interfaz de los mismos (por ejemplo, el nombre, los parámetros que debe poseer, que tipo de dato debe retornar, etc).

Cuando los frameworks son desarrollados en entornos orientados a objetos, generalmente consisten en clases abstractas y clases concretas, donde las concretas son las que se encuentran bajo la clasificación de "frozen spot", debiendo mantenerse intactas, y las abstractas son aquellas que el usuario debe extender para definir código personalizado.

Entonces, una instanciación de un framework puede ser vista como el compuesto entre su código base y los espacios vacíos, que el programador debe llenar:

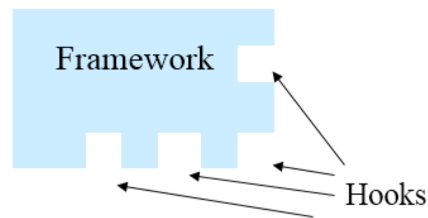

Figura 18: Arquitectura de un framework

Una vez que el desarrollador implementa los hooks methods, el framework es instanciado utilizando a estos, dando origen a nuevas aplicaciones.

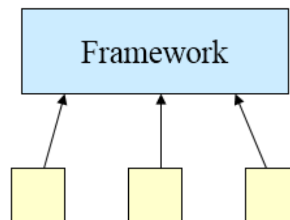

Figura 19: Framework instanciado

2.9 Revisión del objetivo

Ya detallados los diferentes conceptos teóricos, y teniendo estos en cuenta, se revisará en este apartado el objetivo del trabajo.

Habiendo visto qué es un lenguaje de programación visual, las diferentes familias o paradigmas y sus aplicaciones más relevantes, puede notarse que se comparten de forma general algunas ventajas y desventajas. Entre las ventajas, se encuentran:

- ✓ Facilidad de comprensión de programas representados gráficamente
- ✓ Rápido aprendizaje por parte del usuario de la parte visual
- ✓ Facilidad de transmitir/comunicar la lógica de un programa/tarea a áreas no técnicas
- ✓ Reusabilidad de bloques con funcionalidad que se repite

- ✓ Facilidad y velocidad de adaptación de esquemas frente a cambios en el dominio del problema para el que un diagrama o programa fue desarrollado

Mientras que, entre las desventajas, se encuentran:

- Débil en expresión, en comparación a lenguajes textuales para procesos complejos
- Para ciertas actividades que requieren de procesamiento complejo, es más sencillo y más familiar para los programadores utilizar lenguajes textuales.
- Para programadores experimentados no resulta una herramienta de gran utilidad
- Cuando se desarrollan bloques customizados, no todos los lenguajes de programación visual permiten realizar llamados a funciones y/o librerías que no son nativas del lenguaje textual provisto

Además, se ha abordado de forma sintética qué son y como están compuestos los frameworks y cuáles son las características deseables que los mismos posean. Entre las más importantes:

- ✓ Que el aprendizaje para su utilización sea sencillo
- ✓ Que sea fácil de integrar
- ✓ Que sea fácil y simple de utilizar y extender
- ✓ Que defina de forma clara y concisa sus “frozen” y sus “hot” spots.

Teniendo en cuenta los aspectos y tópicos definidos hasta el momento, el objetivo puede explicitarse un poco más:

> desarrollar un framework sencillo y extensible, que pueda ser añadido o integrado con un sistema o programa desarrollado de forma textual, combinando e integrando las ventajas de los lenguajes textuales y visuales en un solo entorno.

Se habla de un framework pensado para sistemas en los que se necesita proveer una forma en la que el usuario final pueda alterar la lógica de ejecución de una forma gráfica o visual sencilla, sin requerir de conocimientos en programación.

Este proveerá, dentro de sus frozen spots, toda la lógica necesaria para crear, almacenar, editar y controlar diagramas, mientras que definirá además todo aquello que el programador deberá implementar. El framework manejará solo la lógica, dejando en los programadores la tarea de crear la interfaz visual para hacer uso del mismo, que podrá ser definida de forma libre. Esto quiere decir que el framework podría ser integrado en cualquier tipo de aplicación, ya sea móvil, web, de escritorio, o de cualquier otro tipo.

Se puede enumerar al menos dos tipos de usuarios para los cuales el framework va a ser útil:

1. <u>El programador o equipo de desarrollo convencional de la aplicación</u>: capaz de continuar el desarrollo del proyecto general, y que además estará encargado de integrar y extender

el framework, haciéndolo funcionar dentro de un proyecto, desarrollando la interfaz visual y creando nuevos bloques con nuevas funcionalidades, sin tener que ahondar en la programación del control principal del manejo de diagramas y bloques.

2. El usuario final: quien podrá definir y modificar el comportamiento de un programa o módulo a través del encadenamiento de bloques funcionales, a través de la interfaz visual provista por el/los desarrolladores.

Capítulo 3: Documentación de Flowgramming Framework

3.1 Introducción

En esta sección se desarrolla la documentación del framework desarrollado. El mismo ha sido bautizado como "Flowgramming Framework", que tiene a la primera palabra como resultado de la contracción de las palabras "flow" (flujo) y "programming" (programación), por lo que podría ser entendido como "Framework de Programación de Flujos"

Este capítulo tiene como objetivo documentar y expresar detalladamente todo lo relacionado al uso del framework. Esto es, una guía que permitirá integrar y hacer uso del mismo, así como también explicará cómo puede ser extendido y personalizado. Además se detallarán diferentes características que provee, para que el desarrollador decida cómo integrarlas/configurarlas, en su persecución por cumplir sus objetivos.

A lo largo de la documentación, se presentarán ejemplos del código, así también como capturas de pantalla de una aplicación web de ejemplo desarrollada con Flowgramming Framework, que tiene como objetivo poder clarificar los conceptos de forma visual, y al mismo tiempo demostrar el potencial y las facilidades provistas por el framework.

La vista principal de la aplicación web tiene el siguiente aspecto:

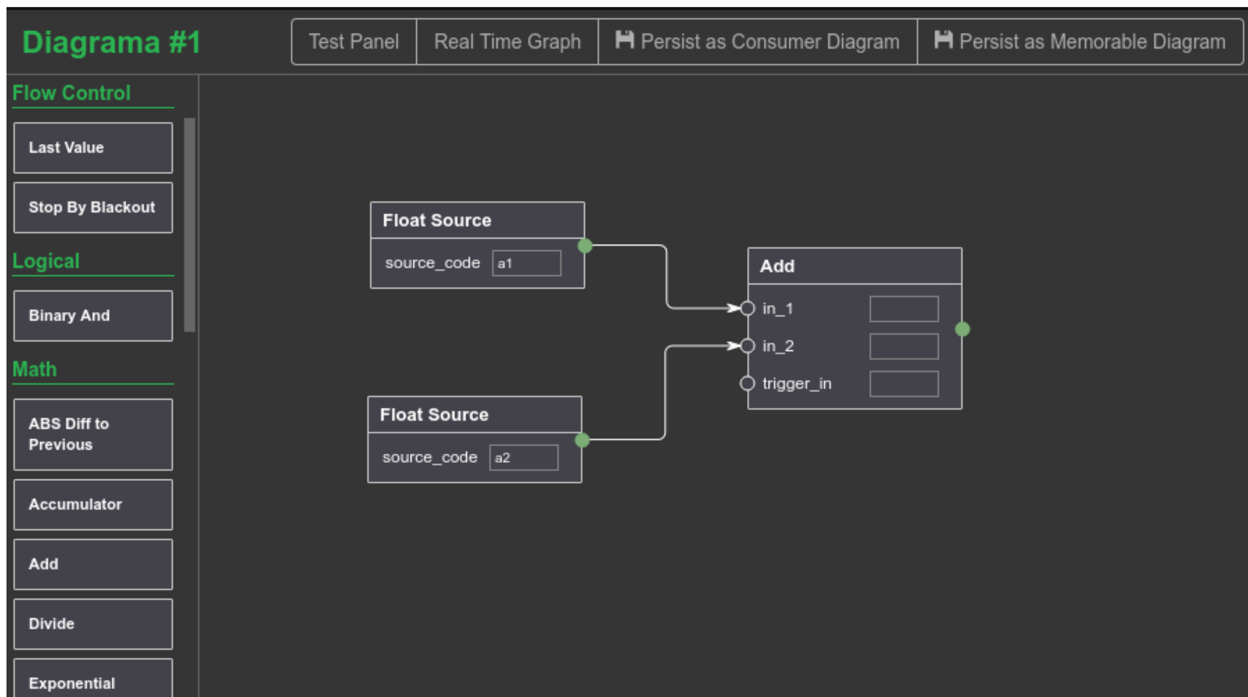

Figura 20: Aplicación de ejemplo de uso de Flowgramming

En la parte superior se encuentran diferentes botones que permiten persistir los diagramas, cambiar el nombre de los mismos y mostrar herramientas de testing. Luego, a la izquierda se encuentra el panel con los bloques disponibles para ser utilizados en los diagramas, mientras que

en el panel principal ubicado en el centro-derecha, se encuentra el área para la generación y edición de los diagramas.

3.2 Requerimientos de software

Flowgramming Framework ha sido desarrollado en y para el lenguaje de programación Python. En un principio, el poder ser integrado sólo con aplicaciones hechas en Python es el único requerimiento o limitación que posee, aunque como trabajo a futuro, se propone que el mismo pueda ser utilizado como un servicio externo, habilitando su integración con cualquier lenguaje de programación. Sin embargo, para poder extenderlo, la programación continuará siendo bajo el mismo lenguaje original.

Actualmente, la implementación del framework es compatible con versiones 2.7 de Python.

3.3 Instalación y dependencias del framework

Como para con todos los proyectos realizados en Python, es recomendable que el "deploy" y el desarrollo del proyecto, así como la instalación de las librerías de las cuales hace uso, se realicen sobre un entorno aislado preparado para este proyecto en particular. Disponer de un entorno aislado es probablemente la mejor configuración posible ya que puede ayudar a evitar muchos problemas en el futuro. Se recomienda el uso de la herramienta universal de Python denominada virtualenv[48]. Los detalles de instalación y uso de la misma se pueden obtener en su sitio oficial, especificado en la referencia.

Una vez "activado" el entorno de Python, simplemente se debe acceder al directorio del sistema donde se encuentra el código del framework, e instalar el mismo a través del archivo "setup.py", con el siguiente comando:

- python setup.py install

Al ejecutar este comando, se instalará el framework y todos los paquetes requeridos para que el mismo funcione de la manera correcta. Una vez que se completó la instalación, ya se puede comenzar a trabajar con el framework.

3.4 Primeros pasos con Flowgramming Framework

3.4.1 Integración

Para comenzar la integración, el primer paso necesario es generar una instancia de Flowgramming Framework, a la cual se hará referencia como "aplicación", o "app" por conveniencia. Esta "app" será el punto de entrada esencial para interactuar con el framework,

por lo que debe ser localizada dentro de un archivo del cual el resto de la aplicación pueda importarla y por lo tanto hacer uso de ella.

Se recomienda agregar un nuevo módulo Python en la estructura del proyecto principal, el cual contendrá todo lo relacionado al uso de Flowgramming Framework. Se puede llamar a este módulo "flowgramming", e incluir un archivo denominado "app.py" en el cuál se definirá la instancia de Flowgramming Framework, dando origen a una estructura similar a la siguiente:

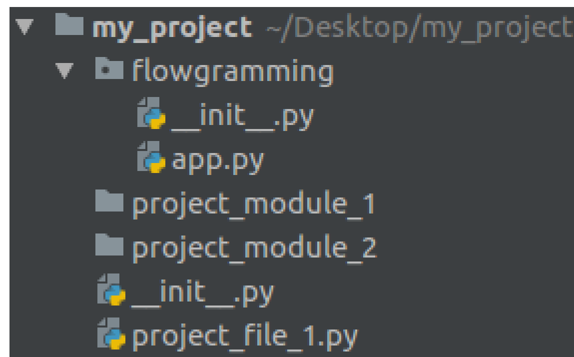

Figura 21: estructura sugerida para el uso de Flowgramming Framework dentro de un proyecto

Luego, para definir nuestra instancia del framework, simplemente, dentro de app.py, se debe incluir el siguiente código:

```
from flowgramming_framework import FlowgrammingApp

app = FlowgrammingApp(
    'CachedFileHashedPersistenceBackend',
    [
        "path_to_my_project/flowgramming_persistence_file.txt"
    ],
)
```

Figura 22: Código básico necesario para instanciar Flowgramming

Una vez creado este archivo dentro del proyecto, la aplicación de Flowgramming Framework ya se encuentra instanciada y lista para ser importada y utilizada desde el resto del proyecto.

Los parámetros de instanciación de este ejemplo, son los únicos estrictamente requeridos. Las opciones siempre se indican con "strings" o listas de "strings". El primero indica la clase controladora (o la clase del "backend") encargada del manejo de persistencia de diagramas a utilizar, mientras que el segundo parámetro es una lista de los parámetros de configuración del backend de persistencia elegido. En este ejemplo se ha seleccionado un backend que persiste los modelos generados en archivos de texto, y que una vez "levantados" o leídos, los almacena/cachea en memoria, y entre los parámetros de configuración, se encuentra, la ruta del destino del archivo que utilizará.

El apartado 3.4.1 tiene como objetivo mostrar de forma simple cómo el framework es instanciado. Más adelante en este capítulo se verán diferentes configuraciones, diferentes backends, y parámetros de configuración avanzada del framework.

3.4.2 Obtener Bloques Disponibles y sus Atributos

Una vez que ya se dispone de una "app", se puede comenzar a construir la interfaz gráfica que hará uso del framework, que funcionará como nexo entre este y el usuario final. El primer paso será el de obtener todos los bloques disponibles para usar en los diagramas, de forma de poder crear un panel con los mismos, del cual el usuario final seleccionará los bloques requeridos para su solución.

Esto se logra llamando al método "get_blocks_data_for_panel" de nuestra app:

```
from flowgramming.app import app

blocks = app.get_blocks_data_for_panel(json=True)
```

Figura 23: Código de obtención de los diferentes tipos de bloques para construir un panel

Para la presente documentación, se utilizará el formato JSON[49], por su claridad y universalidad. Entonces, el método antes mencionado devolverá una estructura similar a la siguiente:

```
[
  {
    "blocks": [
      {
        "output_type": "float",
        "type": "AddBlock",
        "name": "Add",
        "parameters": [
          {
            "default_value": null,
            "disable_default_value": false,
            "input_type": "float",
            "name": "in_1",
            "disable_connector": false
          },
          {
            "default_value": null,
            "disable_default_value": false,
            "input_type": "float",
            "name": "in_2",
            "disable_connector": false
          },
          {
            "default_value": null,
            "disable_default_value": false,
            "input_type": "",
            "name": "trigger_in",
            "disable_connector": false
          }
        ],
        "category_name": "Math"
      },
      ...
    ],
    "category_name": "Math"
  },
  ...
]
```

Figura 24: Resultado (reducido) del método 'get_blocks_data_for_panel'

El resultado de este método, como se puede observar en la figura 24, es, en un principio, una lista de diccionarios, o en términos del formato JSON, una lista de objetos. Los objetos contenidos en esta primera lista representan las categorías de bloques. Luego, para cada categoría, bajo la clave llamada "blocks", se encuentra otra lista de objetos, que representa cada uno de los bloques disponibles para esa categoría.

Entre las claves más importantes para cada objeto que define un bloque se encuentran:

1. **output_type**: El nombre de un tipo primitivo de Python que representa el tipo de salida de un bloque. Al enlazar esta salida con la entrada de otro bloque, los tipos de ambos extremos deben ser iguales.

2. **name**: Nombre para mostrar del bloque, que es independientemente del identificador del tipo o de la clase
3. **type**: Nombre/identificador del tipo/clase del bloque
4. **category_name**: el nombre de la categoría a la cual este bloque pertenece
5. **parameters**: un listado de los parámetros de entrada del bloque, donde para cada parámetro se encuentra la siguiente lista de atributos:
   - **default_value:** El valor por defecto del parámetro al instanciar un bloque
   - **disable_connector:** para algunos parámetros, puede ser que sólo se acepten datos que el usuario ingresa manualmente, por lo que esta opción indica si este parámetro no debe estar conectado con la salida de otro bloque
   - **input_type:** el tipo de entrada del parámetro. Si se conecta este parámetro con la salida de otro bloque, como se ha especificado anteriormente, ambos deben ser del mismo tipo. Si no posee un valor (o si el valor es un String vacío), entonces acepta cualquier tipo de entrada.
   - **disable_default_value**: indica cuando no se debe tomar en cuenta el valor por defecto.
   - **name:** nombre del parámetro.

Cuando el desarrollador programa la parte visual, siempre debe tener en cuenta todas las opciones de sus parámetros y sus bloques, denegando al usuario crear enlaces entre entradas y salidas de diferente tipo, o prohibiendo crear enlaces con parámetros que tienen su conector deshabilitado. De otra forma, si se le permitiera al usuario realizar configuraciones no válidas, a la hora de persistir un diagrama, primero se verifica que este sea válido y es en este punto que no pasará la validación con éxito, retornando los mensajes de error pertinentes.

La información provista por el método mencionado es suficiente para:

1. Crear un panel con todos los bloques disponibles, agrupados por categorías como se muestra en la sección izquierda de la figura 20.
2. Instanciar un bloque de un determinado tipo una vez que el usuario lo elige para utilizarlo en un diagrama. Para seguir con el ejemplo, se muestra a continuación un bloque de Suma (Add), creado a partir de la información provista por la figura 24:

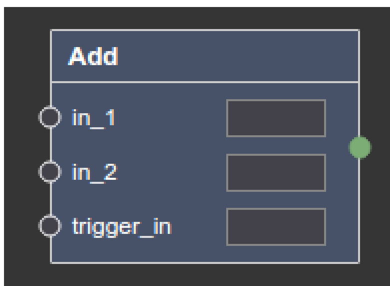

Figura 25:Instancia de un bloque de Suma (Add)

En la aplicación web de ejemplo que se utiliza en este trabajo, la estructura de los bloques será siempre similar a la de la figura 25: en la parte superior se encuentra el nombre del bloque, y dentro del bloque se lista cada uno de sus parámetros, con un conector de entrada a la izquierda de cada uno de ellos (salvo que la definición del parámetro lo prohíba/deshabilite), y a la derecha del bloque, en verde, el conector de salida del mismo.

A modo ilustrativo, se puede decir que una categoría de bloques de ejemplo que poseen parámetros sin conector de entrada son los de tipo "Source". Estos bloques son los que definen el punto de entrada de cada diagrama. Es decir, cuando el sistema general envíe un dato bajo una clave hacia el framework, estos son los bloques buscados para comenzar la ejecución de un diagrama. Haciendo una analogía con los lenguajes de programación textual, estos bloques representan a las variables. Para estos bloques se requiere que el usuario indique la clave para la cual el bloque recibe datos, y este no es un parámetro que deba conectarse con la salida de ningún otro bloque.

Por ejemplo, para un bloque que representa una entrada de un dato de tipo punto flotante, se presenta a continuación la estructura JSON del bloque, y su representación gráfica.

```
{
  "output_type": "float",
  "type": "SourceFloatBlock",
  "name": "Float Source",
  "parameters": [
    {
      "default_value": null,
      "disable_default_value": false,
      "input_type": "str",
      "name": "source_code",
      "disable_connector": true
    }
  ],
  "category_name": "Utils"
}
```

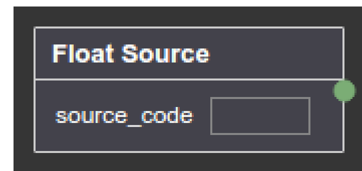

Figura 26: Representación de un bloque de entrada de punto flotante

Como se puede observar en el formato JSON, el conector del parámetro se encuentra deshabilitado, y esto se ve reflejado en la representación visual del bloque.

Al instanciar un bloque, se debe almacenar en alguna estructura, la información perteneciente al mismo, como por ejemplo, cuál es su tipo, cuáles son sus parámetros, etc. Pero, además, un importante atributo a crear al momento de instanciar el bloque es su identificador, que puede ser representado como una cadena de caracteres alfanuméricos. Este va a permitir distinguir cada bloque dentro de un diagrama al momento de persistirlo. En las secciones siguientes se mostrará cómo se hace uso del identificador.

3.4.3 Conectar bloques

Ahora que ya se disponen de los conocimientos necesarios para obtener los tipos de bloques disponibles y como instanciar los mismos para que sean utilizados en un diagrama, el próximo paso es el de desarrollar la forma visual de generar enlaces entre las salidas de los bloques y las entradas de sus parámetros.

A continuación, se muestra un simple ejemplo en el que se quiere sumar, así misma, una entrada identificada bajo la clave x1. Por lo tanto, una de las soluciones puede ser la de instanciar un bloque de tipo "Source" y un bloque de tipo "Suma", donde la salida del bloque de tipo Source, se conecta con cada entrada del bloque de suma:

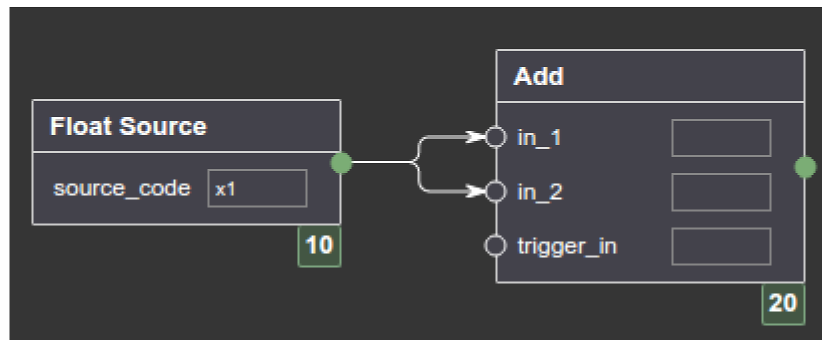

Figura 27: Diagrama sencillo de ejemplo

Al generar enlaces entre bloques y parámetros, hay algunos aspectos importantes a tener en cuenta:

- ✓ Como ya se ha indicado, el tipo de salida del bloque origen debe ser el mismo que el tipo de entrada del parámetro del bloque objetivo (excepto cuando el parámetro de entrada acepta todo tipo de datos)
- ✓ Un bloque puede contener múltiples enlaces conectados a su salida
- ✓ Un parámetro de entrada puede tener a lo sumo un enlace conectado
- ✓ Para cada enlace, se debe almacenar (o al menos se debe poder obtener la información al momento de persistir el diagrama) cuál es el bloque origen, y cuál es el bloque objetivo y el parámetro enlazado del mismo.

Analizando un poco más el último inciso, se puede definir un enlace como una tupla compuesta de la siguiente forma:

enlace = (id_bloque_origen, id_bloque_objetivo, nombre_del_parámetro_objetivo)

Para un enlace, el bloque origen es aquel del cual los datos son expulsados, es decir, el bloque que genera nueva información, o la parte izquierda del enlace. El bloque objetivo, en cambio, es el bloque al que se quiere enviar la salida del bloque origen y su entrada se compone con dos elementos: el identificador del bloque, y el parámetro al cuál se liga el enlace.

3.4.4 Persistir Diagramas

Ahora que ya se sabe cómo se deben representar los bloques instanciados y también sus conexiones, y también habiendo visto los detalles a tener en cuenta para ambas estructuras, el próximo paso en la integración del framework es el de persistir los diagramas, lo que tiene al menos dos objetivos:

1. Que el framework, al recibir un dato correspondiente a una clave, pueda identificar y ejecutar los diagramas que se relacionan con esa clave
2. Que una vez persistidos, los diagramas puedan ser recuperados para poder ser editados

Para poder persistir un diagrama, lo primero que se necesita es una forma de representar el mismo. Un diagrama puede ser visto como un conjunto de bloques y las relaciones existentes entre ellos. Esto se asemeja mucho al concepto de un grafo:

> "En matemáticas y ciencias de la computación, un grafo (del griego grafos: dibujo, imagen) es un conjunto de objetos llamados vértices o nodos unidos por enlaces llamados aristas o arcos, que permiten representar relaciones binarias entre elementos de un conjunto." [35]

Más específicamente, en nuestro caso, podría hacerse referencia los grafos dirigidos, ya que los enlaces de los diagramas de Flowgramming Framework poseen dirección. Una de las diferencias con el concepto de grafo es que los enlaces o aristas no relacionan simplemente nodos o bloques, sino que un enlace relaciona la salida de un bloque con un parámetro de entrada de otro.

Para un diagrama, además de almacenar la información de los bloques y sus relaciones, Flowgramming almacena algunos datos adicionales:

- ✓ Identificador del diagrama
- ✓ Nombre del diagrama
- ✓ Estado: esto es, si el diagrama se encuentra habilitado o deshabilitado
- ✓ Modo de ejecución: Flowgramming provee diferentes modos de ejecución. Este tópico se detalla con mayor profundidad en el apartado 3.5.

Para persistir un diagrama, nuestra "app" dispone una función denominada "persist_diagram", de la cual se muestra, a continuación, un ejemplo de su uso.

Tomando como ejemplo el diagrama de la figura 27, el método persist_diagram, debería ser llamado de la siguiente manera para que el mismo sea persistido:

```python
from flowgramming.app import app

diagram_id = "test_1"
name = "Example"
execution_mode = "C"
enabled = True
blocks = [
    {
        'name': 'Float Source',
        'parameters': [{'default_value': 'x1', 'name': 'source_code'}],
        'position_x': 49,
        'position_y': 81,
        'type': 'SourceFloatBlock',
        'id': '80964b63-b875-4444-8cd1-57e7b7629192'
    },
    {
        'name': 'Add',
        'parameters': [
            {'default_value': None, 'name': 'in_1'},
            {'default_value': None, 'name': 'in_2'},
            {'default_value': None, 'name': 'trigger_in'}
        ],
        'position_x': 330,
        'position_y': 55,
        'type': 'AddBlock',
        'id': 'f2b5fdb4-eca1-41cd-b30b-4a47367170f6'
    }
]
connections = [
    ['80964b63-b875-4444-8cd1-57e7b7629192', 'f2b5fdb4-eca1-41cd-b30b-4a47367170f6', 'in_1'],
    ['80964b63-b875-4444-8cd1-57e7b7629192', 'f2b5fdb4-eca1-41cd-b30b-4a47367170f6', 'in_2']
]


persisted_successfully, errors = app.persist_diagram(
    diagram_id,
    blocks,
    connections,
    name,
    execution_mode,
    enabled
)
```

Figura 28: Ejemplo de persistencia de un diagrama en Flowgramming

Los parámetros utilizados en la función son:

- ✓ **diagram_id**: El identificador bajo el cual se desea almacenar el diagrama. Si se provee un identificador ya existente, el diagrama que se encuentra almacenado bajo ese identificador será reemplazado por el nuevo.
- ✓ **blocks**: una lista de diccionarios Python, donde cada diccionario define un bloque con sus atributos.
- ✓ **connections**: una lista de listas, donde cada lista interna define una conexión de la forma explicada en la sección 3.4.3.
- ✓ **name**: el nombre que se le quiere dar al diagrama.

- ✓ **execution_mode**: el modo de ejecución (este tema se abordará en la sección 3.5).
- ✓ **enabled**: si el diagrama se almacenará habilitado o deshabilitado.

Para este ejemplo, los identificadores de los bloques son generados al azar, pero cada desarrollador podría darles la forma que quisiera. Cada identificador refiere a un bloque dentro de un diagrama, por lo que es válido tener el mismo identificador para bloques distintos definidos en diferentes diagramas.

Además, aparecen en este ejemplo dos nuevos atributos para los bloques: "position_x" y "position_y". Estos atributos no son obligatorios, pero de especificarse, definen la posición del bloque dentro del diagrama en la interfaz visual.

Se puede observar que el método "persist_diagram" devuelve un par de valores. El primero de ellos es un booleano que indica si la persistencia se pudo llevar a cabo con éxito, mientras que el segundo indica los errores en el caso de que los hubiera. Este último parámetro es un diccionario de la forma:

```
{
  "invalid_blocks_definitions_ids": [],
  "invalid_connections": [],
  "critical_cycles_connections_paths": [],
  "non_critical_cycles_connections_paths": []
}
```

Donde cada clave representa un tipo de error, y asociada a cada una de ellas, se encuentra una lista de los elementos que poseen errores. La clave "invalid_blocks_definitions_ids" es una lista con los identificadores de los bloques que se encuentran mal especificados (por ejemplo, un bloque que no tiene conectada una entrada requerida), mientras que "invalid_connections" indica una lista de las conexiones erróneas (donde por ejemplo los tipos de datos a cada extremo de las conexiones no son válidos). En ambos casos, se podrían resaltar en la interfaz visual los elementos erróneos para que el usuario pueda corregirlos. Sin embargo, la lógica presente en la interfaz visual debería validar todos estos elementos de antemano, evitando que el usuario llegue a esta instancia (por ejemplo, no dejando al usuario crear relaciones inválidas). Los errores de tipo "critical_cycles_connections_paths" y "non_critical_cycles_connections_paths" serán explicados con detalle en la sección 3.7.

3.4.5 Obtener Diagramas persistidos

Puesto que ya se sabe cómo almacenar los diagramas, el próximo paso es el de obtener nuevamente la representación de los diagramas almacenados, para que los mismos puedan ser analizados y editados.

Para obtener diagramas, existen dos funciones principales. La primera, denominada “get_persisted_diagrams_data”, permite devolver la información (no la representación) de cada uno de los diagramas almacenados, como por ejemplo, una lista con los identificadores y nombres de todos los diagramas. La segunda, denominada “retrieve_diagram”, devuelve los datos y la estructura de un diagrama en particular.

La primera función es útil para desarrollar interfaces en la que se puedan listar los diferentes diagramas y sus principales atributos. Por ejemplo, si el usuario ha desarrollado 3 diagramas, cuando la función es llamada de la siguiente manera:

```
from flowgramming.app import app

app.get_persisited_diagrams_data()
```

Figura 29: Obtención de datos de diagramas persistidos

Devolverá una estructura similar a:


```
[
  {
    "execution_mode": "C",
    "enabled": true,
    "id": "b9ac934c",
    "name": "Diagram 1"
  },
  {
    "execution_mode": "M",
    "enabled": true,
    "id": "302b18f1",
    "name": "Diagram 2"
  },
  {
    "execution_mode": "C",
    "enabled": false,
    "id": "36ba089b",
    "name": "Diagram 3"
  }
]
```


Figura 30: Salida de ejemplo de datos de diagramas persistidos

Con esta información, se podría generar una interfaz visual como la de la herramienta de ejemplo, en la que el usuario puede seleccionar o editar diagramas que se encuentran en una tabla:

Stored Diagrams

Name	Execution Mode	Enabled	Actions
Diagram 1	Consumer	ON	Edit Delete
Diagram 2	Memorable	ON	Edit Delete
Diagram 3	Consumer	OFF	Edit Delete

Figura 31: Tabla de diagramas persistidos

Luego, la segunda función, “retrieve_diagram”, que recibe el identificador de un diagrama, permite obtener información más detallada y la estructura del mismo, similar a la usada para persistir el diagrama, y a partir de la cual es posible regenerar el diagrama de forma visual, permitiendo al usuario generar modificaciones y almacenarlo nuevamente.

Tomando nuevamente el ejemplo de la figura 27 que fue almacenado en la sección anterior bajo el identificador “test_1”, haciendo uso de la función de la siguiente manera:

```
from flowgramming.app import app

app.retrieve_diagram("test_1")
```

Figura 32: Código de obtención de un diagrama

El resultado será una estructura similar a:

```json
{
  "connections": [
    [
      "80964b63-b875-4444-8cd1-57e7b7629192",
      "f2b5fdb4-eca1-41cd-b30b-4a47367170f6",
      "in_1"
    ],
    ...
  ],
  "blocks": [
    {
      "name": "Float Source",
      "parameters": [
        {
          "default_value": null,
          "name": "source_code",
          "input_type": "str",
          "current_value": "x1",
          "disable_default_value": false,
          "disable_connector": true
        }
      ],
      "position_x": 49,
      "position_y": 81,
      "output_type": "float",
      "type": "SourceFloatBlock",
      "id": "80964b63-b875-4444-8cd1-57e7b7629192"
    },
    ...
  ],
  "data": {
    "execution_mode": "C",
    "enabled": true,
    "id": "b9ac934c",
    "name": "Diagram 1"
  }
}
```

Figura 33:Ejemplo de estructura devuelta por Flowgramming para un diagrama

Como se puede observar, esta estructura es muy similar a la utilizada para almacenar el diagrama, y es suficiente para la regeneración del diagrama de forma visual, que debería quedar igual que en la figura 27, y permitirá al usuario analizar el diagrama y realizar modificaciones si fuera necesario.

3.4.6 Enviar datos a Flowgramming

En este punto, el usuario ya posee las herramientas para crear y editar diagramas. La parte pendiente es la integración de los datos del sistema que hace uso de Flowgramming con el framework. Para ello, existe una función denominada “receive_source_data”, la cual se debe utilizar de la siguiente manera:

```python
from flowgramming.app import app

app.receive_source_data(source, value)
```

Figura 34: Código de ejemplo de envío de datos a Flowgramming

Esta función representa la conexión de datos entre la aplicación general y el framework. Puede ser llamada desde cualquier parte de la aplicación general en la que se quiera enviar datos a Flowgramming. Sus parámetros son:

- ✓ **source:** representa el nombre de la clave para la cual se envía un valor
- ✓ **value:** el valor/dato que se envía

El framework, al recibir un valor proveniente de esta función, buscará todos los diagramas que posean bloques de tipo “Source” donde la clave sea la especificada, y enviará una copia del valor para cada uno de los bloques, comenzando una nueva instancia de ejecución para cada uno de los diagramas.

Luego, es probable que el resultado de la ejecución de un diagrama requiera ser enviado nuevamente a la aplicación que hace uso del framework. Como la forma de recepción de los datos va a variar para cada proyecto particular, este tema se abordará en la sección 3.10.6, que indica cómo generar nuevos bloques para enviar los resultados de nuevo a la aplicación que hace uso de Flowgramming.

3.4.7 Resumen

El propósito de la sección 3.4 ha sido el de detallar los primeros pasos para una integración básica del framework. A lo largo de la misma, se ha desarrollado cómo integrar los principales puntos: cómo obtener bloques, cómo instanciarlos, cómo persistir y obtener diagramas persistidos y cómo enviar datos al framework.

Si bien esta sección detalló generalizadamente cómo integrar el framework, existen muchos detalles que todavía faltan definir. Esto incluye, entre otras cosas, diferentes modos de ejecución del framework y de los bloques, aspectos de ciclos, configuraciones avanzadas y extensión de los módulos principales. Todos estos detalles se detallan en las restantes secciones del capítulo 3.

3.5 Modos de ejecución en Flowgramming Framework

Los diagramas que se ejecutan en Flowgramming Framework admiten dos modos de ejecución, que definen qué se hace con las entradas de los bloques a medida que los mismos se van ejecutando. Estos modos son:

1. “Consumer” o modo de ejecución consumidora
2. “Memorable” o modo de ejecución con memoria

Cuando un diagrama comienza a ejecutarse, los datos comienzan a fluir desde las salidas de los bloques hacia las entradas de los otros bloques a través de sus conexiones. Cada bloque se va a ir ejecutando tan pronto como reciba nuevos datos en sus entradas y disponga de todos los datos necesarios para ejecutar su operación.

La aplicación web de ejemplo representa cómo un dato fluye por un enlace coloreando el mismo de verde. Tomando un caso de ejemplo en el que simplemente se quieren multiplicar dos datos, bajos las claves 'x' e 'y' se puede obtener el siguiente diagrama:

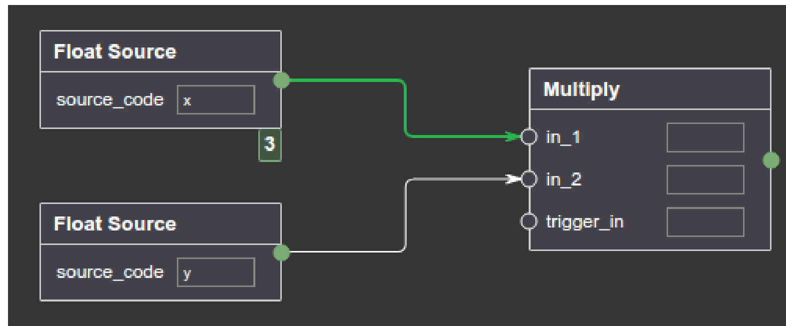

Figura 35: Ejemplo de flujo de datos en un enlace

La figura 35 representa un estado de ejecución en el cual se puede observar que se ha enviado un dato para la clave 'x', pero para la clave 'y' todavía no ha enviado ningún dato. Luego, en nuestro escenario de ejemplo, 'y' está próximo a enviar un dato, por lo que el bloque de multiplicación podrá ser ejecutado. Este momento, incluyendo la ejecución del bloque de multiplicación, puede verse de la siguiente manera:

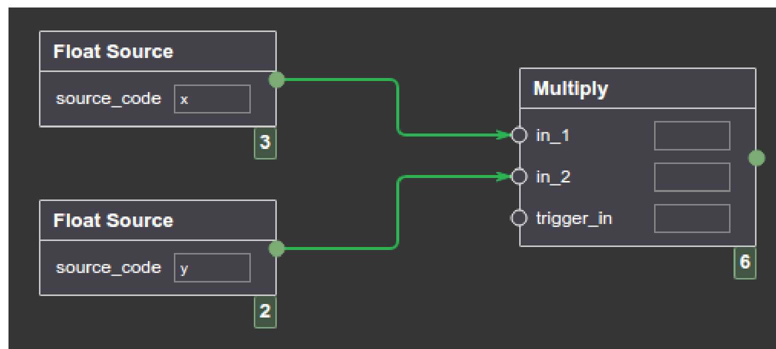

Figura 36:: Ejemplo de flujo de datos #2

Hasta este punto, cualquiera de los modos de ejecución funcionaría de la misma forma. Lo que altera cada uno de los modos es lo que sucede con las entradas de un bloque luego de que este ejecuta su operación.

En este momento posterior a la ejecución de un bloque, se debe definir qué hacer con las entradas del mismo. Existen dos alternativas:

1. Se podría “limpiar”, “consumir” o “borrar” los datos de cada una de las entradas, y tener que recibir nuevamente datos por todas las entradas para ejecutar el bloque nuevamente. Este es el modo de ejecución “**consumer**”.
2. Se podrían conservar las entradas, y ejecutar el bloque cada vez que se reciba un nuevo dato por alguna (cualquiera) de sus entradas. Así es como funciona el modo “**memorable**”.

A continuación se analizarán dos casos de ejemplo para los pasos siguientes a la figura 36, uno para cada uno de los modos. Comenzando primero con el método “memorable”, luego de que se ejecuta, el estado del diagrama queda igual que en la figura 36. Luego, suponiendo que se envía un nuevo valor para clave ‘y’, entonces el diagrama se ejecuta, quedando en el siguiente estado:

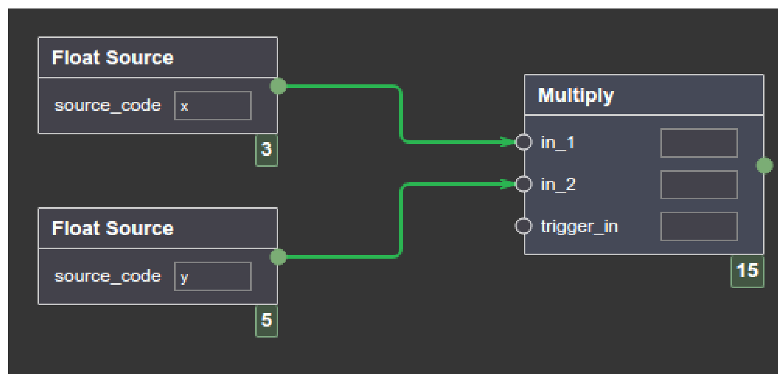

Figura 37: Ejemplo de flujo de datos en modo Memorable

Como se puede observar, el estado anterior de la primera entrada del bloque de multiplicación, correspondiente al dato ‘x’, se ha conservado y tan pronto como ha llegado un nuevo valor para ‘y’, el bloque de multiplicación se vuelve a ejecutar.

Ahora, volviendo al estado de la figura 36, si se considera el modo de ejecución “consumer”, antes de que se continúe con la ejecución de nuevos datos, el framework “limpia” las entradas del bloque de multiplicación, quedando el estado del diagrama de la siguiente forma (notar el color blanco de sus enlaces):

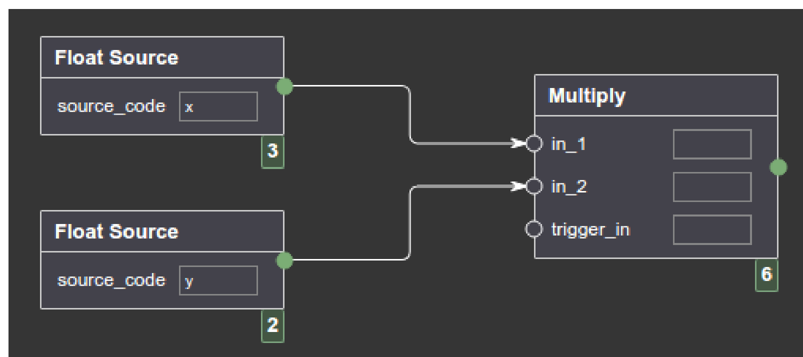

Figura 38: Ejemplo de flujo de datos en modo Memorable #1

Por lo tanto, para que el bloque de multiplicación se vuelva a ejecutar es necesario que vuelva a recibir datos por ambas entradas nuevamente, como al comienzo de la ejecución. Primero recibirá un dato por una de sus entradas:

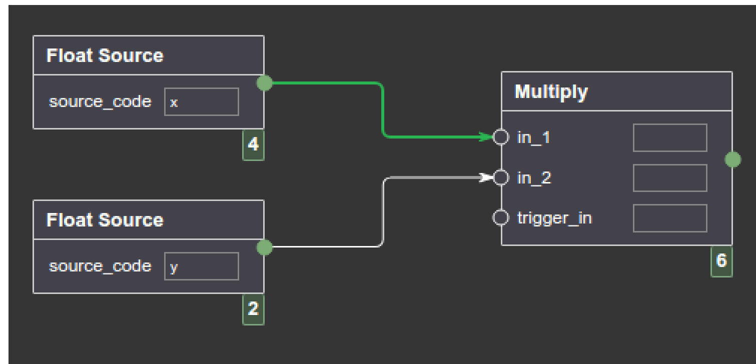

Figura 39: Ejemplo de flujo de datos en modo Memorable #2

Pero debido a que el modo de ejecución eliminó los datos de sus entradas luego de la ejecución previa, todavía necesita recibir un dato nuevamente por la segunda entrada para que ejecute la operación de multiplicación nuevamente. Suponiendo que 'y' envía ahora un nuevo dato, entonces se ejecuta la operación de multiplicación:

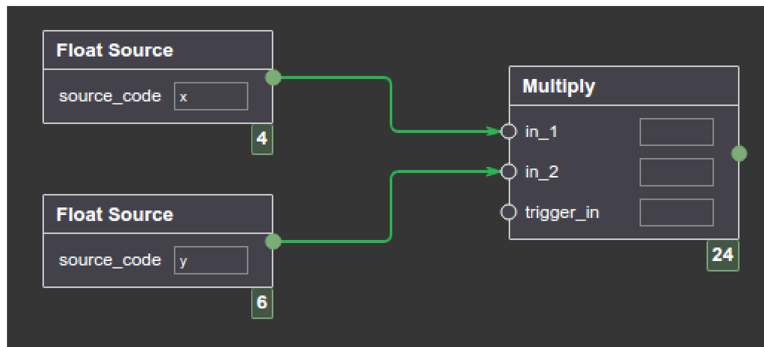

Figura 40: Ejemplo de flujo de datos en modo Memorable #3

Y luego se vuelven a limpiar las entradas:

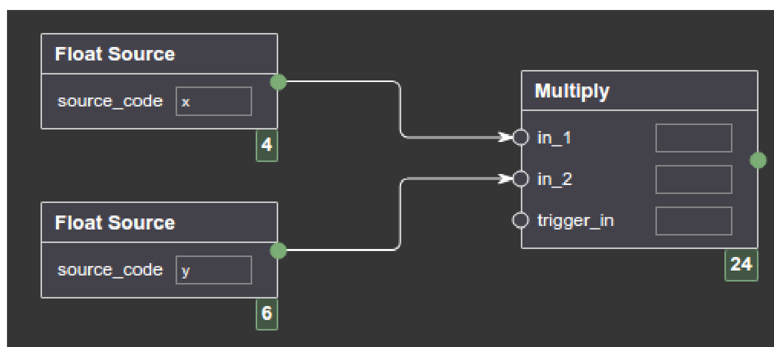

Figura 41: Ejemplo de flujo de datos en modo Memorable #4

Ahora que se describieron ambos modos de ejecución, se indica a continuación cómo seleccionar cada uno. Para persistir un diagrama, como se mencionó, se utiliza la función "persist_diagram". El modo de ejecución está definido por el quinto parámetro posicional de la misma, siendo el carácter "C" el correspondiente al modo consumidor, y el carácter "M" al modo memorable:

```
execution_mode = "C" | "M"
persisted_successfully, errors = app.persist_diagram(
    diagram_id,
    blocks,
    connections,
    name,
    execution_mode,
    enabled
)
```

Figura 42: Persistencia de un diagrama en diferentes modos de ejecución

Un último aspecto a tener en cuenta en cuanto a la ejecución de los diagramas es el aspecto no-determinístico de la misma. Al tener un diagrama con múltiples bloques y múltiples conexiones para sus salidas, no se puede determinar, a priori, cuál será el orden en el que los datos fluirán por las relaciones, y, por lo tanto, qué bloque se ejecutará a continuación. Los bloques se ejecutarán con un orden no-determinístico tan pronto como estén listos para ejecutar una nueva operación, de acuerdo a las reglas definidas por cada uno de los modos de ejecución.

3.6 Modos de ejecución y condiciones de los bloques

Además de los modos de ejecución a nivel diagrama definidos en el anterior apartado, existen otros aspectos a nivel de los bloques que alteran la ejecución de los diagramas. Quizás a lo largo de la documentación, el lector pudo notar que casi todos los bloques, con excepción de los de tipo "Source", poseen un parámetro adicional bajo el nombre "trigger_in" (esto puede notarse tanto en la descripción textual como visual de los bloques). Este parámetro, cuando está conectado, define una condición de ejecución adicional para un bloque en particular.

Un bloque que posee este parámetro conectado se ejecutará cuando se cumplan dos condiciones:

1. Cuando todas sus entradas estén listas para que el bloque sea ejecutado (según las reglas del modo de ejecución seleccionado) y,
2. Cuando se haya recibido un dato por esta entrada (trigger_in)

Es decir, que además de las condiciones que pone cada uno de los modos de ejecución, para que el bloque sea ejecutado debe recibir algún dato por este parámetro, que es el que lo habilita para que pueda ser ejecutado. Una vez que el bloque es ejecutado, la entrada de este parámetro

siempre se limpia independientemente del modo de ejecución, siendo necesario recibir un dato por la misma entrada nuevamente para que el bloque se vuelva a ejecutar.

Es importante notar que esta condición adicional funciona a nivel bloque, por lo que podría dar un aspecto de ejecución particular a un área de un diagrama, independientemente del modo de ejecución seleccionado y de cómo se ejecuta el resto del diagrama.

El tipo de dato que esta entrada acepta es indiferente, siendo válido cualquier tipo. Además, si el bloque recibe muchos datos para sus otros parámetros antes de que esta entrada especial reciba algún dato, al momento de ejecutarse, el bloque se ejecutará con los últimos datos recibidos para cada una de sus entradas. De modo inverso, si se reciben muchos datos por este parámetro adicional pero no por los otros, el bloque no se ejecuta hasta que no disponga de todos los otros parámetros necesarios; y de nuevo, una vez ejecutado se limpia, es decir, no es acumulativo.

Para clarificar este punto, se desarrolla a continuación un ejemplo gráfico. Se supone para el ejemplo que se quiere restar la entrada de dos variables 'x' e 'y'. Además, 'x' e 'y' mandan valores constantemente al framework, pero sólo se desea realizar el cálculo frente a la ocurrencia de un evento 'z'. Entonces se podría generar el siguiente diagrama, que se ejecutará bajo el modo "consumer":

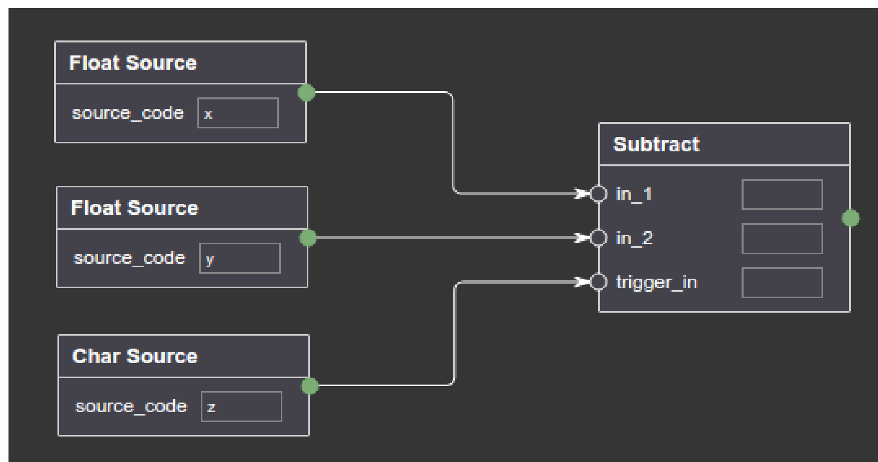

Figura 43: Ejemplo de ejecución condicional: estado #1

Ahora, 'x' e 'y' mandan su primer conjunto de valores:

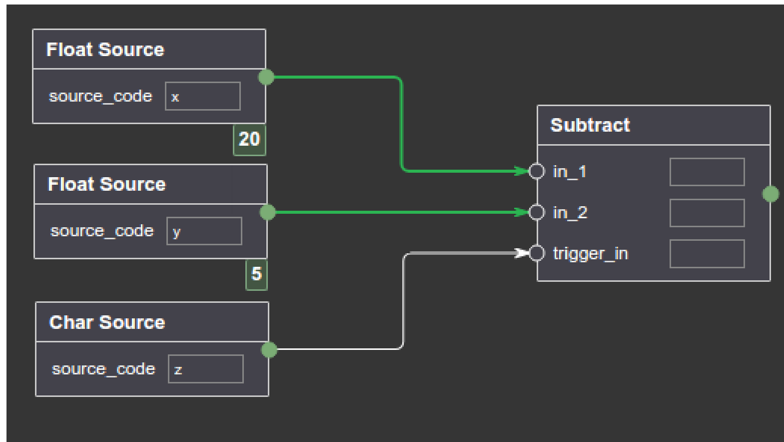

Figura 44: Ejemplo de ejecución condicional: estado #2

Pero como el evento 'z' todavía no ha ocurrido, no se procede con la ejecución del bloque. 'x' e 'y' continúan mandando nuevos datos

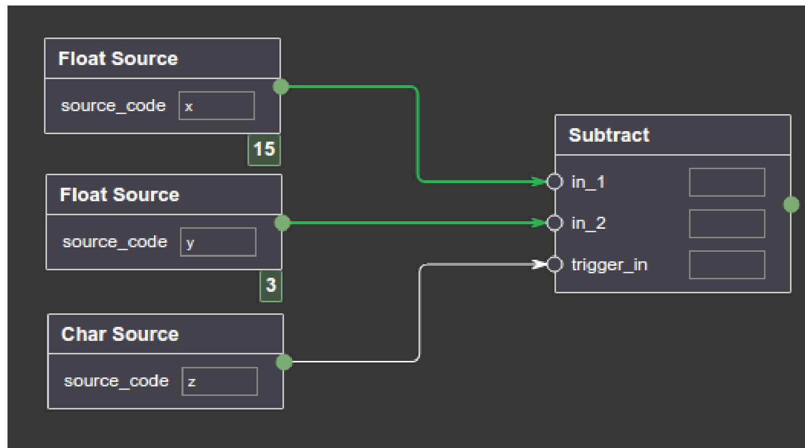

Figura 45: Ejemplo de ejecución condicional: estado #3

Y luego ocurre el evento 'z', dando lugar a que el bloque de resta ejecute su operación, con los últimos valores recibidos para "x" e "y":

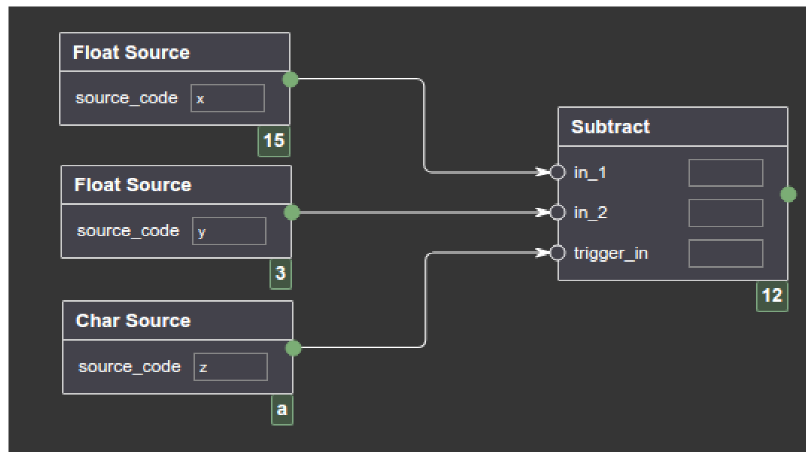

Figura 46:: Ejemplo de ejecución condicional: estado #4

Esta funcionalidad de ejecución condicional, además de ser útil para la señalización de eventos como se ha mostrado en el anterior ejemplo, puede ser de mucha utilidad frente a la ocurrencia de ciclos, para evitar que los mismos resulten en errores. Los diferentes tipos de ciclos y la forma en la que la ejecución condicional puede modificarlos es el tema del siguiente apartado.

3.7 Ciclos

3.7.1 Tipos de ciclos

A la hora de diseñar diagramas, teniendo en cuenta tanto los modos de ejecución que el framework provee como la ejecución condicional a nivel de los bloques, se debe tener especial cuidado con los ciclos.

La presencia de ciclos podría ser crítica, generando que el diagrama ejecute una secuencia de operaciones infinitas, recorriendo los bloques de un cierto ciclo dentro de un diagrama. Esto, en un principio tiene un primer error crítico: si el framework ejecuta ese ciclo sin parar, el resto del diagrama no se ejecuta nunca. Pero este no es el único error. Existe otro segundo problema relacionado con la actual implementación, ya que el framework se ejecuta en un solo thread o hilo de ejecución, por lo que, si se encontrara ejecutando los bloques de un ciclo, además de no ejecutar otras partes del diagrama al cual el ciclo pertenece, no ejecutará tampoco otros diagramas.

La presencia de ciclos es más probable en diagramas complejos que en diagramas que representan flujos simples, y se supone que cuanto más complejo sea un diagrama, probablemente el diseñador del mismo sea una persona más especializada. El tratamiento y la consideración de ciclos debe ser controlado por quien diseña los diagramas teniendo en cuenta lo dicho hasta el momento y considerando los diferentes modos de ejecución y las conexiones del diagrama. Se podría hacer una analogía con los lenguajes de programación convencionales y la ocurrencia de iteraciones infinitas: es el programador el encargado de que este escenario no

suceda (excepto que se busque ese comportamiento adrede), es decir, el encargado de evitar que un programa se quede iterando infinitamente.

Sin embargo, aunque el tratamiento de ciclos debería ser considerado por quien diseña un diagrama, el framework provee facilidades e indicadores de los mismos, aunque no se garantiza una ejecución exitosa, ya que esto depende mucho de las condiciones dadas en tiempo de ejecución.

Cuando se intenta persistir un diagrama, Flowgramming evalúa la presencia de ciclos, clasificándolos en dos categorías:

1. Ciclos críticos
2. Ciclos no críticos

Se considera ciclo **crítico** a aquel que, al presentarse en un diagrama, generará una ejecución infinita de operaciones, independientemente de las condiciones de ejecución, o del valor que los campos puedan tomar. Un ciclo **no crítico** será aquel en el cuál la presencia de una ejecución infinita va a depender de las condiciones que se den durante la ejecución, condiciones que el diseñador del diagrama podría controlar.

Esta clasificación, además, dependerá del modo de ejecución del diagrama. Un mismo ciclo podría ser crítico cuando el diagrama se ejecute en modo “memorable” pero no crítico si el diagrama se ejecuta en modo “consumer”.

A continuación se presentan algunos ejemplos que permitirán clarificar las situaciones. En la aplicación web, luego de intentar persistir un diagrama, los ciclos críticos son resaltados en rojo y los no críticos en amarillo.

Suponiendo que se quiere persistir un diagrama como el siguiente:

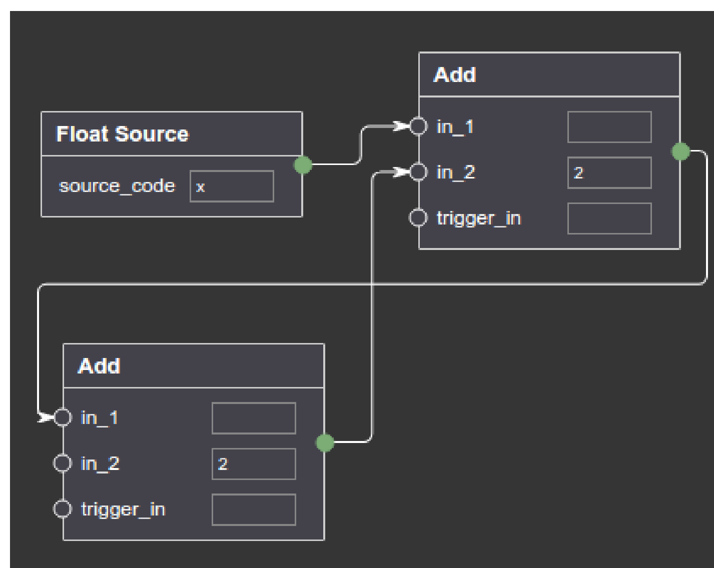

Figura 47: Ejemplo de diagrama con ciclos

En este diagrama se puede notar que existe un ciclo, ya que la salida del primer bloque de suma (el de arriba) se dirige al segundo, y la salida de este último se dirige a la entrada del primer nuevamente, reiniciando el bucle. Si se quisiera persistir este diagrama en modo “memorable”, es decir, almacenando el estado de las entradas y ejecutando los bloques cada vez que se recibe una nueva entrada de datos, entonces es evidente que el ciclo sería crítico, ya que la salida de cada bloque de suma iniciaría la ejecución del otro. Esto es lo que en Flowgramming se conoce como un error crítico, y podría ser resaltado para que el usuario lo corrija, como se muestra a continuación:

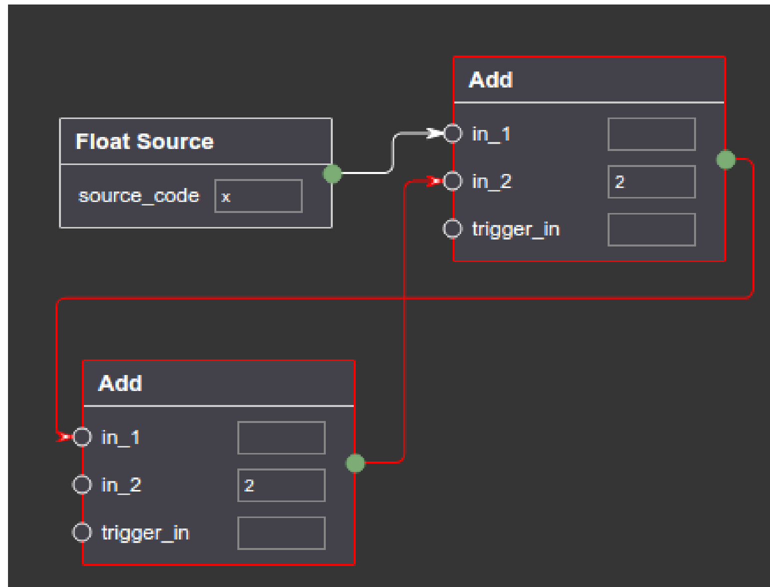

Figura 48: Ejemplo de ciclo crítico

Ahora, si se quisiera persistir el diagrama en modo “consumer”, entonces las entradas se borrarían luego de que un bloque sea ejecutado. A priori, en este ejemplo sencillo se puede observar que este ciclo, almacenado bajo este modo de ejecución no tendría aparejado ningún inconveniente. Sin embargo, en diagramas más complejos, con muchos bloques y conexiones, esto sería más difícil de ver y de identificar; por esto, Flowgramming notifica de todas formas al usuario, diciéndole de alguna forma “cuidado, hay un bucle”, que, si bien no es crítico, el hecho de que se genere o no un error es responsabilidad del diseñador del mismo, quien tendrá que tener en cuenta las posibles condiciones de ejecución. La ocurrencia de ciclos no críticos podría representarse con otro color más “suave”, como se muestra a continuación:

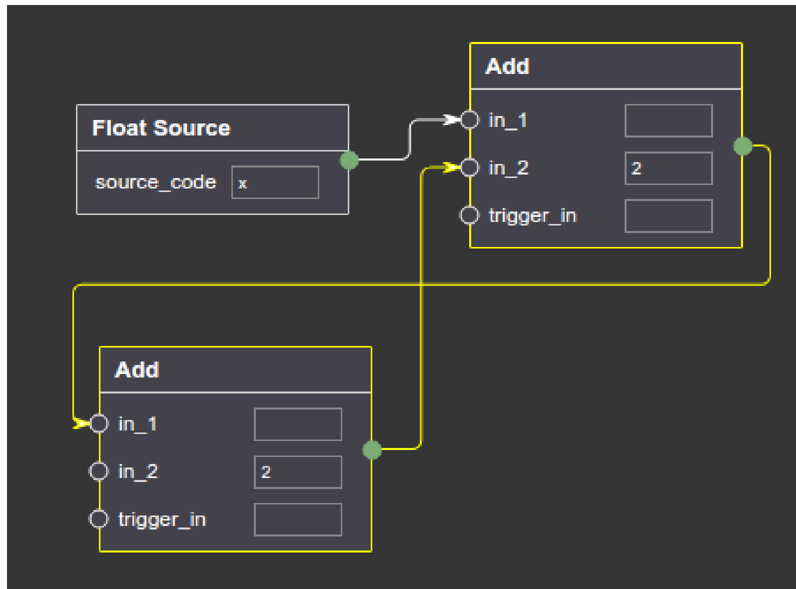

Figura 49: Ejemplo de ciclo no crítico

Además de los modos de ejecución, otro factor que podría alterar la clasificación de un mismo ciclo en crítico o no crítico es la ejecución condicional presentada en el anterior apartado. Volviendo al primer caso mostrado en la figura 47, en el que se presenta un ciclo crítico bajo el modo "memorable", el mismo podría dejar de ser crítico con una pequeña modificación, si la ejecución de alguno de los bloques fuera condicional, haciendo uso del parámetro "trigger_in". Por ejemplo, si se altera el diagrama de la siguiente forma:

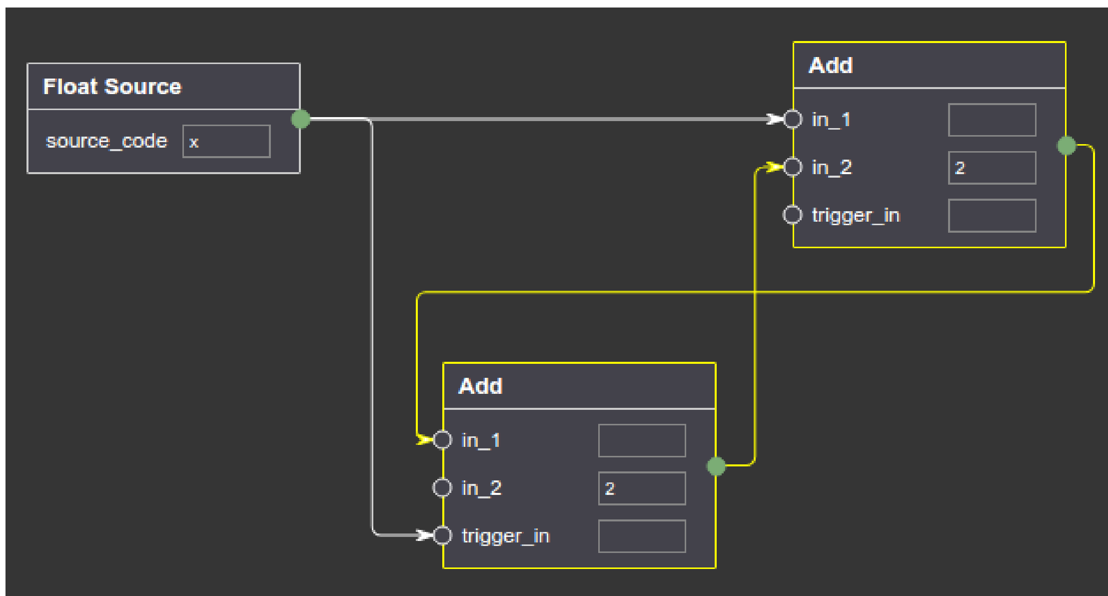

Figura 50: Conversión de ciclo crítico en no crítico

entonces el ciclo dejará de ser crítico, ya que, para cada valor recibido de "x", se permitirá solo una ejecución completa del ciclo. De nuevo, en este caso esto puede ser visto fácilmente, pero

se le indica al usuario que podría haber algún tipo de problema, ya que dependiendo del estado de las conexiones y de los valores de los datos al momento de la ejecución de un diagrama más complejo, esto podría resultar en un error “fatal”. Básicamente, una de las reglas para definir si un ciclo es crítico o no bajo el modo de ejecución “memorable”, es verificar si alguno de los bloques que componen el ciclo poseen ejecución condicional o no.

Por lo tanto, como se mostró en los anteriores ejemplos, la opción de ejecución condicional puede ser una útil herramienta para abordar los ciclos críticos.

3.7.2 Validación de diagramas e identificación de ciclos

Como se ha mencionado en las secciones anteriores, Flowgramming antes de persistir un diagrama realiza diferentes tipos de validaciones para verificar que todo lo relacionado al diagrama está correctamente definido. Es en este paso en el cual también verifica la existencia de ciclos.

Frente a la ocurrencia de un ciclo (y suponiendo en este caso que no hay ningún otro tipo de error) el framework persistirá correctamente un diagrama siempre que no posea ciclos críticos. Caso contrario, es decir, en caso de que exista un ciclo crítico, no se persistirá el diagrama hasta que el ciclo sea corregido.

Se presenta a continuación el resultado de los errores de la función “persist_diagram” (que ya se ha presentado previamente), para un diagrama que posee un ciclo no crítico:

```
{
    "invalid_connections": [],
    "non_critical_cycles_connections_paths": [
        [
            (
                "5b8bd933-4292-4b96-9aa9-7affdd328d72",
                "32e88a51-7706-485c-9558-1cdd700dd25b",
                "in_1"
            ),
            (
                "32e88a51-7706-485c-9558-1cdd700dd25b",
                "5b8bd933-4292-4b96-9aa9-7affdd328d72",
                "in_2"
            )
        ]
    ],
    "invalid_blocks_definitions_ids": [],
    "critical_cycles_connections_paths": []
}
```

Figura 51: Ejemplo de resultado de verificación de un diagrama con errores

Recordar que la función devuelve un par de valores, donde uno indica si el diagrama ha sido persistido correctamente y el otro indica errores (si los hubiera). Como se puede observar, la clave "non_critical_cycles_connections_paths” es una lista de los ciclos que el diagrama presenta, donde cada ciclo, a su vez, está representado como una lista de tuplas, donde cada una representa una conexión que conforma el ciclo.

En caso de existir un ciclo crítico la respuesta es similar, donde la diferencia radica en que el listado de ciclos se encuentra bajo la clave "critical_cycles_connections_paths”.

Además, la función “persist_diagram” posee un parámetro posicional bajo el nombre “disable_if_non_critical_cycles_present”, que es un booleano que indica si un diagrama que posee ciclos no críticos debe ser automáticamente deshabilitado luego de ser persistido. Con esta funcionalidad, se le podría mostrar un mensaje al usuario indicando que el diagrama ha sido satisfactoriamente persistido, pero que, debido a la ocurrencia de un ciclo no crítico, debe ser habilitado manualmente.

3.7.3 Seguimiento de errores en ciclos en tiempo de ejecución

Como se ha indicado, para los diagramas que presentan ciclos críticos, el framework no puede identificar si van a causar o no errores en tiempo de ejecución, y esto es algo que el diseñador del diagrama debería considerar para minimizar la ocurrencia de errores tanto como sea posible.

Sin embargo, el framework puede identificar la ocurrencia de ciclos que tienden a ser infinitos en tiempo de ejecución. En las próximas secciones se detallará cómo el framework notifica errores, pero por ahora basta decir que, frente a la ocurrencia del evento en cuestión, lo notificará junto con el identificador del diagrama que lo ha producido.

3.8 Configuración del framework

3.8.1 Instanciación del framework

Como se ha descripto previamente, para hacer uso del framework es necesario crear una “instancia” del mismo con la cual interactuar. Para ello, el framework provee la clase “FlowgrammingApp”, que al ser instanciada provee un punto de conexión con el framework.

El framework permite diferentes tipos de configuraciones y diferentes backends de soporte para que se pueda ejecutar sobre diferentes tipos de recursos y en diferentes modos. Para indicar la configuración deseada, la inicialización de esta clase admite diferentes parámetros. Entre ellos, se encuentran los primeros dos que son obligatorios, y que en el orden correspondiente indican:

1. El nombre de la clase del backend de persistencia a utilizar, es decir, el nombre de la clase que manejará la forma en la que los diagramas con sus bloques y relaciones son almacenados y la forma de obtener/recuperar los mismos.

2. Una lista con los diferentes parámetros de configuración para la clase previamente definida, donde esta lista de parámetros es definida por la clase seleccionada. Por ejemplo, si se seleccionara un backend de persistencia en archivos, es aquí donde se indicaría que archivo utilizar. Si, en cambio, se seleccionara un backend orientado a bases de datos, aquí se podrían indicar parámetros como el “host” de la base de datos, el nombre de la misma, nombre de usuario y contraseña.

Luego, se dispone de una serie de parámetros opcionales (del tipo “keyword” de Python) que poseen valores por defecto. Entre estos, se encuentran:

- ✓ “error_logging_backend_class_name": la clase que se encargará del logging de errores ocurridos durante tiempo de ejecución en un cierto diagrama. Valor por defecto: 'ConsoleErrorLoggingBackend'
- ✓ “error_logging_backend_settings”: de forma análoga a los parámetros de configuración del framework de persistencia, esta es una lista con los parámetros opcionales de configuración del backend de logging de errores. El valor por defecto es una lista vacía.
- ✓ “custom_blocks_paths”: una lista de strings, donde cada string representa un camino o ruta hacia un archivo, donde cada uno de ellos contendrá tipos de bloques personalizados o propios del desarrollador que hace uso del framework. El valor por defecto de este parámetro es una lista vacía.
- ✓ “debugger_backend_class_name”: indica el nombre de una clase que será utilizada para notificar los resultados de la ejecución de cada bloque cuando se está haciendo una simulación con datos de prueba, para poder notificar a la aplicación y que el usuario vea, en tiempo real, los resultados de cada bloque para los datos de simulación enviados. Por defecto, no se encuentra definido ningún backend de este tipo.
- ✓ “debugger_backend_settings”: una lista con los parámetros de configuración del backend de “debug” definido por el anterior parámetro.

Los diferentes backends provistos tanto para la notificación de errores en tiempo de ejecución, como para los de datos de simulación, son temas que se abordarán con mayor detalle en los siguientes apartados.

Los parámetros de configuración del framework admitirán strings para indicar el nombre de clases de backends y listas de strings para indicar parámetros de configuración de cada uno de los backends.

Entre las combinaciones de configuración que podrían ser relevantes para el conocimiento del usuario, se encuentran diferentes casos. El primer caso es aquel en el que el usuario configura el framework con backends o clases provistas, como por ejemplo:

```
app = FlowgrammingApp(
    'CachedFileHashedPersistenceBackend',
    ["path_to_my_project/flowgramming_persistence_file.txt"],
)
```

Figura 52: Ejemplo de instanciación de Flowgramming con backend de persistencia específico

Como se hace uso de una clase provista por el framework, el nombre de la clase es suficiente. Pero si, en cambio, el usuario quisiera utilizar el framework con un backend de persistencia definido por él, entonces es necesario que al nombre de la clase se le agregue el camino o ruta hacia el archivo que lo contiene, relativa al proyecto que hace uso de Flowgramming. El framework se encargará de encontrar la clase del backend especificado. Por ejemplo:

```
from flowgramming_framework import FlowgrammingApp

app = FlowgrammingApp(
    'my_project.flowgramming.custom_persistence_backends.CustomPersistenceBackend',
    [""],
)
```

Figura 53: Ejemplo de instanciación de Flowgramming con backend de persistencia "personalizado"

3.8.2 Backends de Persistencia

Como se ha indicado previamente, los backends de persistencia son los que se encargan del manejo de persistencia de los diagramas con sus respectivos bloques y conexiones y los atributos propios de ellos. El framework incluye 2 tipos básicos de backends de persistencia: DummyMemoryHashedPersistenceBackend y CachedFileHashedPersistenceBackend.

El primero sólo almacena los diagramas con sus modelos en memoria. Si bien no es útil para la ejecución del framework de forma real, permite realizar verificaciones rápidas de diagramas y testear si el framework y su configuración funcionan de manera correcta. Este backend no requiere de parámetros de inicialización.

El segundo backend, un poco más concreto, persiste los diagramas en archivos de texto, y al obtener los diagramas, los almacena en memoria de modo que los posee "cacheados", lo que en tiempo de ejecución provee una mejora de eficiencia a la hora de recuperar bloques de un diagrama, frente a la idea de tener que leer los bloques siempre desde archivos, ya que las operaciones de lectura de discos son mucho más costosas que las de memoria. Este backend requiere un solo parámetro de inicialización, que indica la ruta absoluta del archivo en el cuál se persiste la información relevante a los bloques.

Ambos backends son provistos al usuario para que el mismo pueda ver a grandes rasgos cómo se deben implementar concretamente, pero el desarrollador debería crear los suyos propios, de la forma que más le convenga.

3.8.3 Backends de Debugging

Cuando un usuario está diseñando un diagrama, es muy probable que quiera enviar datos de prueba para analizar si las salidas de cada bloque son las esperadas.

Por ejemplo, en la aplicación de ejemplo provista para este trabajo, se puede observar el caso en el que un usuario desea sumar los datos asociados a las claves "x" e "y", y para esto hace uso de una herramienta que permite enviar datos de prueba (Test Panel):

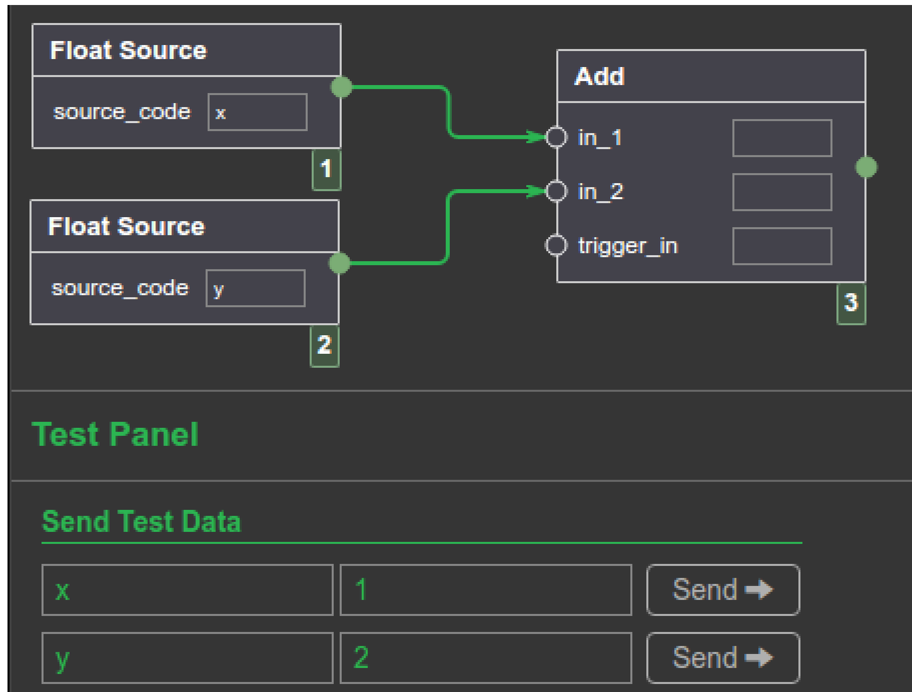

Figura 54: Envío y notificación de datos de prueba

En la esquina inferior izquierda de cada bloque, se puede ver la salida de cada uno para los datos de prueba enviados.

Para llevar a cabo una lógica similar a la presente en la figura, se necesita que el framework provea las facilidades para enviar datos de prueba y además para notificar las salidas de cada uno de los bloques.

Para enviar datos de prueba, estos se envían al framework de la misma forma que se ha indicado previamente para los que no son de prueba (esto es, haciendo uso de la función "receive_source_data"). Pero para este caso se le adicionará un parámetro opcional denominado "is_testing_execution" con el valor booleano "True". Al indicar este parámetro con este valor, el framework sabe que para le ejecución de cada bloque debe notificar la salida.

Para la notificación de resultados de los bloques, el framework provee un sencillo backend, con el nombre "DummyMemoryOutputValuesDebuggerBackend". Este backend no posee

parámetros de inicialización, y básicamente almacena en memoria los valores de los bloques, hasta que se le piden, momento en que los devuelve y retorna el resultado.

La forma de obtener los resultados de los bloques es la siguiente:

```
app.debugger_backend.get_blocks_outputs()
```

Que devuelve como resultado, una lista con los resultados de cada bloque, donde cada resultado a su vez se representa como una tupla que sigue el siguiente formato:

```
(id_diagrama, id_bloque, valor_de_la_salida)
```

Este backend debe utilizarse con cuidado, debido a que por su naturaleza de almacenar resultados en memoria, si se enviaran muchos datos sin pedirle los resultados de los bloques, el backend podría utilizar una considerable cantidad de memoria. Por este motivo su uso debiera limitarse a casos muy básicos, y se recomienda la generación de backends propios por parte del desarrollador con alguna técnica de publish/subscribe o pasaje de mensajes, en la que no se almacenen los resultados de forma temporal.

El motivo por el cual el framework no provee diferentes tipos de backends de debugging, es porque cada usuario tendrá requerimientos muy diferentes, y esto se traduciría en distintas implementaciones que harían uso de diversas librerías que serían requeridas para la instalación base del framework. La política es que el desarrollador escriba sus propios backends, utilizando las librerías necesarias de acuerdo a sus recursos y requerimientos.

3.8.4 Backends de logging de errores

El último aspecto importante de la configuración es el que permite el seguimiento de errores que han ocurrido en tiempo de ejecución. Para cumplir con tal objetivo, el framework provee dos backends: “ConsoleErrorLoggingBackend” y “FileErrorLoggingBackend”.

El primero de ellos es el establecido por defecto, no requiere de parámetros de inicialización y es el que muestra los errores ocurridos en tiempo de ejecución en la consola o terminal utilizada para ejecutar la aplicación que hace uso de Flowgramming. Este backend es útil para entornos de desarrollo.

Para entornos de producción, en los que generalmente las aplicaciones corren como servicios, es de mayor utilidad persistir los errores en archivos de texto que luego los desarrolladores pueden examinar. Para ello se utiliza la clase FileErrorLoggingBackend. Este backend requiere un solo parámetro de inicialización que es la ruta hacia el archivo en el cuál se almacenarán los errores. Un ejemplo de instanciación del framework que usa este backend se muestra a continuación:

```python
from flowgramming_framework import FlowgrammingApp

app = FlowgrammingApp(
    'CachedFileHashedPersistenceBackend',
    ["path_to_my_project/flowgramming_persistence_file.txt"],
    error_logging_backend_class_name='FileErrorLoggingBackend',
    error_logging_backend_settings=['/var/log/flowgramming/flowgramming_errors.txt']
)
```

Figura 55: Ejemplo de instanciación de Flowgramming con backend de logging de errores específico

3.9 Bloques Provistos por el Framework

En esta sección se listarán los bloques que el framework provee por defecto junto a sus parámetros de entrada y una breve explicación de su funcionalidad. En la sección siguiente se abordará la forma de crear nuevos bloques.

Los bloques del framework tienen como objetivo brindar funcionalidades básicas. Las operaciones específicas para cada proyecto serán provistas por los desarrolladores a través de la generación de bloques personalizados.

Se listan a continuación los bloques provistos por el framework, agrupados por categorías:

Nombre del Bloque	Parámetros/ Entradas	Descripción	Tipo de salida
Categoría: Casting			
Boolean To Char	input (bool)	Toma una entrada booleana y la convierte en una cadena de caracteres	char
Boolean To Float	input (bool)	Toma una entrada booleana y la convierte en un número de punto flotante, donde 1.0 representa el valor “True” y 0.0 representa el valor “False”	float
Boolean To Int	input (bool)	Toma una entrada booleana y la convierte en un número de tipo entero, donde 1 representa el valor “True” y 0 representa el valor “False”	int
Char To Bool	input (char)	Toma una entrada que es una cadena de caracteres y la convierte en un booleano, siguiendo las reglas que Python indica para este tipo de conversión	bool

Char To Float	input (char)	Toma una cadena de caracteres que representa un número y la convierte en un número de tipo punto flotante	float
Char To Int	input (char)	Toma una cadena de caracteres que representa un número y la convierte en un número de tipo entero	int
Float to Bool	input (float)	Toma una entrada que es un número de tipo punto flotante y la convierte en un booleano, siguiendo las reglas que Python indica para este tipo de conversión	bool
Float to Char	input (float)	Toma una entrada que es un número de tipo punto flotante y la convierte en una cadena de caracteres	char
Float to Int	input (float)	Toma una entrada que es un número de tipo punto flotante y la convierte en número de tipo entero	int
Int to Bool	input (int)	Toma una entrada que es un número de tipo entero y la convierte en un booleano, siguiendo las reglas que Python indica para este tipo de conversión	bool
Int to Char	input (int)	Toma una entrada que es un número de tipo entero y la convierte en una cadena de caracteres	char
Int to Float	input (int)	Toma una entrada que es un número de tipo entero y la convierte en un número de punto flotante	float
Categoría: Control/Conditional			
Binary And	in_1 (float), in_2 (float)	Dadas dos entradas de tipo flotante, que deben representar a los números 1 o 0, realiza una operación lógica AND, donde el resultado también se expresa como un número de punto flotante	float
Binary Not	Input(float)	Dada una entrada de tipo punto flotante, donde la misma deberá representar un 1 o un 0, realiza la operación lógica NOT, donde el resultado también se expresa como un número de tipo flotante	float
Binary Or	in_1 (float), in_2 (float)	Dadas dos entradas de tipo punto flotante, que deben representar a los	float

		números 1 o 0, realiza una operación lógica OR, donde el resultado también se expresa como un número de punto flotante	
Condition to Boolean	Input (float), condition(char)	Dadas dos entradas, donde una representa un número de tipo flotante, el bloque permite definir una condición en su otra entrada, evaluándola en base al valor recibido y expulsando el resultado como un valor de tipo booleano	bool
Conditional Stop Execution	Input (float), condition(char)	Dadas dos entradas, donde una representa un número de tipo punto flotante, permite definir una condición en su otra entrada. Si la condición es verdadera, la ejecución se frena en el bloque. De otra forma, el bloque expulsa el mismo dato que recibió por su entrada “input”	float
Float Same Value Stop Execution	input(float)	Dada una entrada, el bloque detiene la ejecución en el bloque cuando la misma recibe valores iguales de forma consecutiva	float
Greater Stopper	input (float), threshold (float)	Dada una entrada de tipo punto flotante, la compara con una referencia (threshold) y detiene la ejecución si el valor recibido es más grande que la referencia	float
Last Value	in_1 (float), in_2 (float)	Dadas dos entradas de tipo punto flotante, envía por la salida el valor de la última entrada recibida	float
Lower Stopper	input (float), threshold (float)	Dada una entrada de tipo punto flotante, la compara con una referencia (threshold) y detiene la ejecución si el valor recibido es más chico que la referencia	float
PeriodicalThrottler	input (float), period_seconds(int)	Dada una entrada de tipo punto flotante, reenvía la misma por la salida solo cuando han pasado al menos una cantidad de segundos especificada	float

Stop Execution	input(bool)	Dada una entrada de tipo booleana, detiene la ejecución en caso que la entrada represente el valor “True”	bool
Category: Math			
ABS Diff to Previous	input(float)	Dada una entrada de tipo punto flotante, envía a la salida la diferencia absoluta de los valores consecutivas que recibe por la misma	float
Accumulator	input(float), reset(any)	Dada una entrada de tipo punto flotante, suma (acumula) los valores que recibe por la misma y envía el resultado de esta acumulación a la salida. Si se le envía algún dato por “reset”, la cuenta se reinicia para la próxima recepción de datos	float
Add	in_1 (float), in_2 (float)	Dadas dos entradas de tipo punto flotante, envía por la salida el valor resultante de la suma de las mismas	float
Divide	dividend (float), divisor (float)	Dadas dos entradas de tipo punto flotante, donde una representa el dividendo y la otra el divisor, realiza la operación de división	float
Exp	input(float)	Dada una entrada de tipo punto flotante, envía a la salida el resultado del número irracional e, potenciado por el número indicado por la entrada	float
Exponential Moving Average	input (float), filter_parameter (float)	Dadas dos entradas de tipo punto flotante, donde una representa una entrada y la otra un parámetro de filtro, que debe ser mayor que 0 y menor que 1, realiza la operación media móvil exponencial	float
Factorial	input(float)	Dada una entrada de tipo punto flotante, envía a la salida el resultado de la operación factorial aplicada a la misma	float
Log	in_1 (float), in_2 (float)	Dada una entrada de tipo punto flotante, realiza la operación logarítmica, aplicando la base expresada por su otra entrada	float

Max	in_1 (float), in_2 (float)	Dadas dos entradas de tipo punto flotante, envía por la salida el valor mayor de ambas	float
Min	in_1 (float), in_2 (float)	Dadas dos entradas de tipo punto flotante, envía por la salida el valor menor de ambas	float
Multiply	in_1 (float), in_2 (float)	Dadas dos entradas de tipo punto flotante, envía por la salida el valor resultante de la multiplicación de las mismas	float
Power	base (float), exponent (float)	Dadas dos entradas de tipo punto flotante, donde una representa la base y otra el exponente, realiza la operación de potencia	float
Random	range_from (float), range_to (float)	Dadas dos entradas de tipo punto flotante, que representan el comienzo y el fin de un rango numérico, envía por la salida un número aleatorio contenido en ese rango	float
Round	input (float), digits (int)	Dada una entrada de tipo punto flotante, redondea la misma a la cantidad de dígitos decimales expresada por su otra entrada	float
SQRT	input(float)	Dada una entrada de tipo punto flotante, envía a la salida el resultado de la operación raíz cuadrada aplicada a la misma	float
Subtract	in_1 (float), in_2 (float)	Dadas dos entradas de tipo punto flotante, envía por la salida el valor resultante de la resta de las mismas	float
Category: Source			
Char Source	source_code (char)	Dada una clave, representa la entrada a un diagrama de un dato enviado al framework por la aplicación que hace uso del mismo, donde el tipo de dato del valor es una cadena de caracteres	char
Float Source	source_code (char)	Dada una clave, representa la entrada a un diagrama de un dato enviado al framework por la aplicación que hace uso del mismo,	float

		donde el tipo de dato del valor es numérico de punto flotante	

3.10 Creación de bloques personalizados

3.10.1 Introducción

Este apartado es quizás uno de las más importantes, ya que describe cómo el desarrollador puede definir nuevos tipos de bloques, lo que implica quizás, la parte más importante del framework.

3.10.2 Estructura del código de un bloque

Siguiendo el lema "un código expresa más que su explicación", valorado por muchos desarrolladores para quienes mayoritariamente es más sencillo primero ver el código para entender cómo algo funciona, lo primero que se mostrará son las líneas de un sencillo bloque que el framework provee, el de suma, y luego se desarrollará su explicación:

```
class AddBlock(GenericBlock):
    output_data_type = float

    in_1 = FloatField(is_input=True)
    in_2 = FloatField(is_input=True)

    class Meta:
        disabled = False
        display_name = "Add"
        category_name = "Math"

    def execute(self):
        return self.in_1 + self.in_2
```

Figura 56: Código del bloque de suma

Lo primero que se puede observar, es que el bloque extiende de una clase abstracta denominada "GenericBlock". Esta clase es la clase base que el framework define, y de la cual todos los bloques deben extender, debido a que implementa las operaciones básicas para que cualquier bloque pueda funcionar, y además sirve para distinguir las clases que definen bloques del framework de otras clases.

Lo segundo que se puede observar, es que posee un atributo denominado "output_data_type". Este atributo define cual es el tipo de salida del bloque, que será útil para la validación de los diagramas que se realiza previo a su persistencia.

Luego se pueden observar dos atributos, denominados in_1 e in_2. Ambos atributos son instancias de una clase denominada "FloatField" y en su inicialización indican que son de tipo "input". Los bloques de Flowgramming funcionan con clases declarativas[3], donde los campos de tipo numérico y de cualquier tipo, son instancias de las clases correspondientes. Esto permite que cada campo provea, además del valor que se recibe por la entrada del mismo durante la ejecución, otros meta-datos adicionales que serán de gran relevancia para la ejecución del framework. En este caso, ambas entradas son de tipo Float, es decir, entradas de datos numéricos de tipo flotante.

También se puede observar que se define una clase interna al bloque con el nombre Meta. Esta clase, cuando está presente, brinda información adicional acerca del bloque. En este caso, indica que el bloque se encuentra habilitado para su uso, así también como el nombre para mostrar y el nombre de la categoría a la cual el bloque pertenece.

Por último, el bloque define un método "execute", que hace uso de los campos previamente definidos y devuelve el resultado de la operación para la cual fue desarrollado.

Si se evitan las líneas de la clase Meta (clase que no es requerida) se puede notar que con tan solo seis líneas de código se pudo definir un bloque completo.

Esta breve cantidad de líneas es suficiente para que el framework, entre otras cosas, pueda:

- ✓ Devolver el bloque y sus atributos cuando se pide la información para renderizar los tipos de bloque
- ✓ Instanciar un bloque para ser utilizado dentro de un diagrama
- ✓ Persistir y recuperar el bloque en tiempo de ejecución
- ✓ Ejecutar la operación que el bloque reciba
- ✓ Validar los diagramas en base a las entradas y las salidas de los bloques

Esta característica de definición de un bloque de una forma tan breve es una de las principales características que maximizan la potencia del framework y la velocidad de desarrollo.

3.10.3 Desarrollo de bloques personalizados

Ya descripta la estructura del código de un bloque, se indica en ese apartado una breve guía acerca de cómo el desarrollador podría diseñar nuevos bloques.

Para el desarrollo de un nuevo bloque, en un módulo Python definido en la aplicación del usuario, el desarrollador debe implementar una clase que extienda de "GenericBlock". Por ejemplo:

[3] Este tema se aborda con mayor detalle en la sección 4.2

```python
from flowgramming_framework import GenericBlock


class MyBlock(GenericBlock):
```

Figura 57: Definición de bloque personalizado. Paso #1

Luego, el desarrollador debe indicar el tipo de salida del bloque, y el nombre y el tipo de los campos de los cuales el framework va a hacer uso:

```python
from flowgramming_framework import GenericBlock
from flowgramming_framework import fields


class MyBlock(GenericBlock):
    output_data_type = float

    input_1 = fields.FloatField(is_input=True)
    input_2 = fields.FloatField(is_input=True)
    parameter = fields.FloatField(is_input=False)
```

Figura 58: Definición de bloque personalizado. Paso #2

Como se puede observar, uno de los parámetros posee un campo del tipo is_input=False, esto indica que este parámetro no debe ser conectado mediante enlaces, pero permite que el usuario indique manualmente un valor para el mismo.

El último paso dentro de las tareas requeridas a la hora de implementar un bloque es el desarrollo de la función que realizará el mismo, retornando el resultado. Esto se indica en el método "execute":

```python
class MyBlock(GenericBlock):
    output_data_type = float

    input_1 = fields.FloatField(is_input=True)
    input_2 = fields.FloatField(is_input=True)
    parameter = fields.FloatField(is_input=False)

    def execute(self):
        return (self.input_1 + self.input_2) * self.parameter
```

Figura 59: Definición de bloque personalizado. Paso #3

En este punto, lo básico y necesario ya se encuentra definido, pero se puede además indicar un nombre (de otra forma se mostraría el nombre de la clase) y la categoría a la cual se desea que el bloque pertenezca:

```python
class MyBlock(GenericBlock):
    output_data_type = float

    input_1 = fields.FloatField(is_input=True)
    input_2 = fields.FloatField(is_input=True)

    class Meta:
        display_name = "My Function"
        category_name = "Custom Blocks"

    def execute(self):
        return (self.input_1 + self.input_2) * self.input_2
```

Figura 60: Definición de bloque personalizado. Paso #4

El último paso es el de indicar la ruta relativa de este archivo para que el framework pueda ir a buscar bloques dentro del mismo. Como se ha indicado previamente, esto se realiza durante la instanciación del framework. Un ejemplo podría ser:

```python
from flowgramming_framework import FlowgrammingApp

app = FlowgrammingApp(
    "CachedFileHashedPersistenceBackend",
    ["path_to_my_project/flowgramming_persistence_file.txt"],
    custom_block_paths=[
        "web_app.flowgramming.custom_blocks",
    ]
)
```

Figura 61: Ejemplo de instanciación de Flowgramming con rutas donde buscar bloques extra

En este punto, el framework ya sabe cómo proveer la información para mostrar el bloque definido, cómo instanciarlo, cómo utilizarlo, etc.

Se puede observar a continuación, como se vería el bloque definido en la aplicación web de ejemplo provista, tanto en el panel de bloques como haciendo uso del mismo en el diagrama, y enviando algunos datos de prueba:

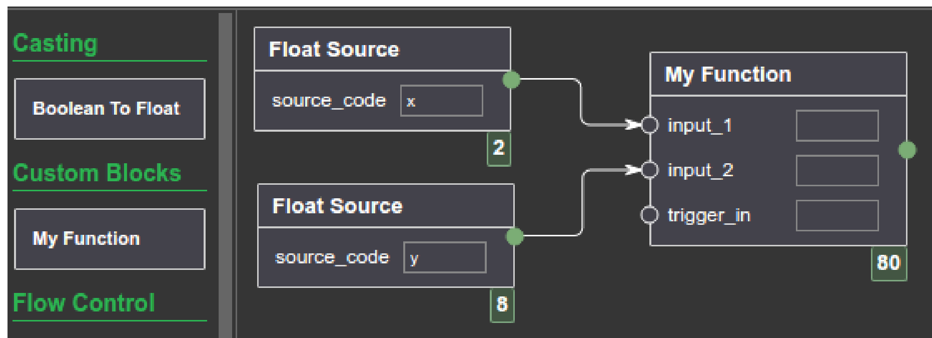

Figura 62:Utilización de bloque personalizado

Además, los bloques permiten definir otro método que sirve para darle un formato específico a la salida cuando el resultado es mostrado a través del backend de debugging. El método recibe el resultado enviado por el método "execute", y debe retornar el valor formateado. Por ejemplo, suponiendo que la salida de un bloque de suma contiene muchos decimales:

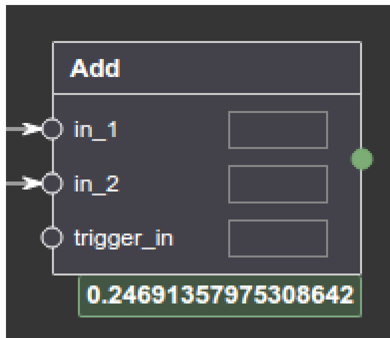

Figura 63: Salida compleja de un bloque

El proceso de desarrollo con datos de prueba se podría volver un poco tedioso si se presentan datos con este formato, o en casos donde los resultados de bloques poseen estructuras más complejas. Para eso existe el método mencionado anteriormente. En este caso, se podría decidir redondear el resultado para que solo se muestren 2 decimales, de la siguiente forma:

```
def get_debugger_output_formatted(self, result_value):
    return round(result_value, 2)
```

Figura 64: Formateo de la salida de un bloque

Y luego, la misma salida tendría el siguiente aspecto:

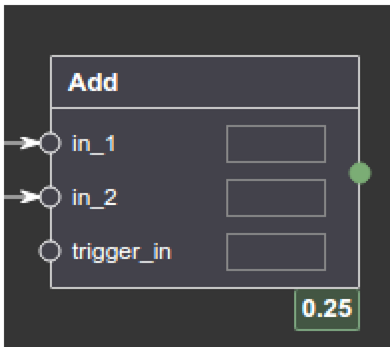

Figura 65: Salida formateada de un bloque

3.10.4 Fields

Los campos representan los datos variables de los cuales los bloques hacen uso al momento de ejecutar la operación para la cual fue definido, por lo que resultan de gran importancia y es por este motivo que se los describe a continuación.

3.10.4.1 Campos Provistos por el Framework

Con el objetivo de poder crear nuevos bloques para diferentes fines de forma flexible, sencilla y cómoda, el framework provee un conjunto de fields de diferentes tipos, así también como la posibilidad de crear nuevos tipos. La creación de fields personalizados está descripta en el apartado 3.10.4.3.

El framework brinda un conjunto de fields que representan los diferentes tipos básicos de datos de cualquier lenguaje de programación. En la siguiente tabla, se muestra cada una de las clases provistas, junto con el tipo de dato que tienen asociado:

Clase	Tipo de Dato	Tipo Primitivo en Python
Charfield	Cadenas de caracteres (Strings)	str
IntegerField	Números de tipo Entero	int
FloatField	Números reales	float
BooleanField	Booleanos	bool
DateTimeField	Tiempo y fecha	datetime

Para la validación de diagramas, al verificar una conexión entre la salida de un bloque y un parámetro de entrada de otro, se utilizará siempre el tipo expresado en la tercera columna para validar que la salida del primer bloque sea compatible con el tipo primitivo del campo que el otro extremo de la conexión tiene asociado. Si el usuario quisiera realizar alguna conexión entre un bloque y un parámetro que no son compatibles, deberá utilizar alguno de los bloques de 'casting' implementados para tal objetivo. En la siguiente figura se muestra un ejemplo en el que un dato de tipo string es convertido a número real y, posteriormente a booleano:

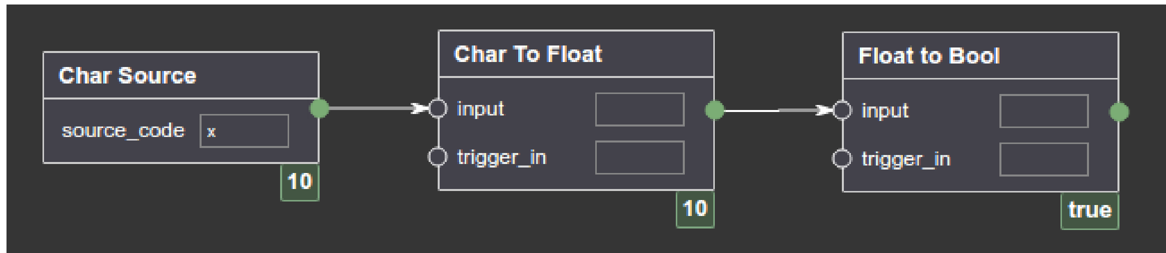

Figura 66: Diagrama con casting de datos

3.10.4.2 Opciones de configuración de fields

Cuando se declara un conjunto de campos que van a ser utilizados por un determinado bloque, estos son instanciados con un conjunto de parámetros de modo que se almacenan meta-datos asociados a cada uno de los campos, utilizados tanto para la ejecución, como en la instanciación de los bloques cuando se diseña un diagrama.

Como se ha visto en los ejemplos anteriores, estas opciones que caracterizan un campo no son estrictamente requeridas, y son indicadas mediante parámetros de tipo “palabra clave” cuando el campo del bloque es declarado.

A continuación se listan los diferentes parámetros con sus posibles valores, y luego se detallará, conjuntamente con ejemplos de declaración y también con su representación gráfica, la explicación y el uso de cada uno de estos parámetros:

Parámetro	Tipo de dato	Valor por defecto
default_frontend_value	El tipo relacionado con el campo	None
is_input	bool	False
disable_default_value	bool	False
disable_connector	bool	False
accept_any_type	bool	False
is_required_input_for_execution	bool	True

Los primeros cuatro parámetros afectan a la parte visual y por lo tanto serán ejemplificados con código y su mapeo a la representación gráfica, mientras que los últimos dos sólo afectan a la lógica de ejecución de los diagramas.

default_frontend_value: este parámetro debe ser tenido en cuenta por el desarrollador de la interfaz visual e indica el valor por defecto que el campo debería tomar inicialmente en caso que el usuario final no lo cambie. A continuación, se puede observar un ejemplo de un bloque con un campo que hace uso de este parámetro, junto con su representación gráfica:

```python
class Example(GenericBlock):
    output_data_type = float

    in_1 = fields.FloatField(default_frontend_value=10.0)
    in_2 = fields.FloatField(default_frontend_value=5.5)

    class Meta:
        display_name = "Example"

    def execute(self):
        return self.in_1 ** self.in_2
```

```
Example
in_1        10
in_2        5.5
trigger_in
```

Figura 67: Uso del parámetro la opción de fields "default_frontened_value"

is_input: existen algunos casos en los que se desea almacenar datos temporales en un bloque, como por ejemplo cálculos intermedios relativos a la funcionalidad para la que un bloque fue desarrollado, o contadores, acumuladores, etc. Estos datos son usados de forma interna por el bloque y no deben ser visibles para el usuario final. Es en este caso que este parámetro toma importancia, ya que su fin es el de indicar si el campo debe ser mostrado o no en la interfaz gráfica.

```python
class Example(GenericBlock):
    output_data_type = float

    in_1 = fields.FloatField(default_frontend_value=10.0)
    accumulated = fields.FloatField(is_input=False)

    class Meta:
        display_name = "Example"
        persist_after_execution = True

    def execute(self):
        self.accumulated += self.in_1
        return self.accumulated
```

```
Example
in_1        10
trigger_in
```

Figura 68: Uso del parámetro la opción de fields "is_input"

disable_default_value: indica cuándo se debe ocultar o cuándo se debe restringir al usuario final de poder insertar valores por defecto, siendo para el caso de la aplicación de ejemplo, esconder el campo de texto, denegando al usuario la chance de poder ingresar algún valor, y que la entrada solo provenga de los enlaces:

```python
class Example(GenericBlock):
    output_data_type = float

    in_1 = fields.FloatField(default_frontend_value=10.0)
    in_2 = fields.FloatField(disable_default_value=True)

    class Meta:
        display_name = "Example"

    def execute(self):
        return self.in_1 + self.in_2
```

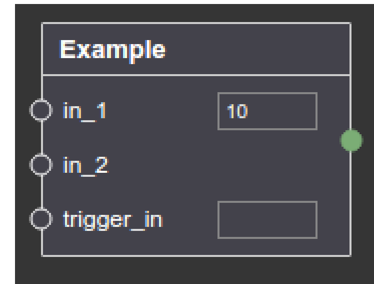

Figura 69: Uso del parámetro la opción de fields "disable_default_value"

disable_connector: para algunos campos se desea que el usuario final ingrese manualmente un determinado valor, pero no se desea que este pueda ser conectado para recibir salidas de otros bloques. Cuando se le asigna un valor "verdadero", se le indica al desarrollador de la interfaz visual que este campo no debe poder ser conectado con la salida de otros bloques. Además, cuando se verifica la validez de un diagrama, se chequea que el parámetro no esté conectado mediante un enlace.

```python
class Example(GenericBlock):
    output_data_type = float

    in_1 = fields.FloatField(default_frontend_value=10.0)
    output_file = fields.FloatField(disable_connector=True)

    class Meta:
        display_name = "Example"

    def execute(self):
        save(self.in_1, self.output_file)
```

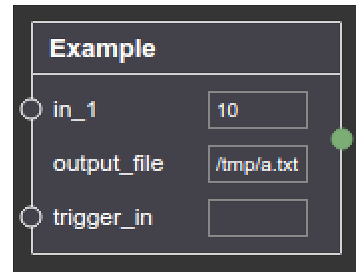

Figura 70: Uso del parámetro la opción de fields "disable_connector"

accept_any_type: cuando se asigna este parámetro en True, independientemente del tipo del campo que se utilice, permite enlazar la salida de cualquier bloque, sin importar su tipo, con la entrada de este parámetro.

is_required_input_for_execution: para algunos bloques, podrían existir datos que, si están presentes, se utilizarán para la ejecución del bloque, pero que no son críticos para la ejecución. Este parámetro, cuando es "verdadero" indica que el dato asociado a este campo no es estrictamente necesario para llevar a cabo la ejecución, por lo tanto al evaluar si un bloque debe ser ejecutado (en tiempo de ejecución), no se tiene en cuenta si este campo posee o no un valor.

3.10.4.3 Desarrollo de fields personalizados

Además de los fields provistos por el framework, el desarrollador puede definir y utilizar sus propios fields que se ajusten de la mejor forma posible a sus requerimientos. La forma de definir un nuevo tipo de field es a través de la creación de una clase que extienda de “BaseField”, que se encuentra bajo el módulo “fields” de Flowgramming.

El único requerimiento necesario para crear un nuevo tipo de campo, es que el tipo de dato asociado a este pueda ser serializado y des-serializado, es decir, que los datos deben poder convertirse en strings, y que además, se provea la forma de obtener el dato desde la representación el mismo almacenado en un string. Para la serialización y des-serialización, el desarrollador debe definir los métodos to_serial y to_python respectivamente.

Se muestra a continuación como el campo “DateTimeField” (utilizado para trabajar con fechas y horas) hace uso de los métodos previamente mencionados:

```
class DateTimeField(BaseField):
    accepted_input_type = datetime.datetime

    def __init__(self, format=None, serial_format=None, **kwargs):
        super(DateTimeField, self).__init__(**kwargs)
        self.format = format
        self.serial_format = serial_format

    def to_python(self):
        if self.data is None:
            return None

        # don't parse data that is already native
        if isinstance(self.data, datetime.datetime):
            return self.data
        elif self.format is None:
            # parse as iso8601
            return PySO8601.parse(self.data)
        else:
            return datetime.datetime.strptime(self.data, self.format)

    def to_serial(self, time_obj):
        if not self.serial_format:
            return time_obj.isoformat()
        return time_obj.strftime(self.serial_format)
```

Figura 71: Código de ejemplo de un campo personalizado

Como se muestra en la figura 71, la clase definida extiende de “BaseField”. Luego, se define el tipo de datos que el campo acepta como entrada, mediante el atributo “accept_input_type”.

La clase define los métodos de instancia “to_python”, que hace uso del atributo de instancia “data”, el cual corresponde al valor almacenado en el campo, y define cómo convertirlo a un objeto de tipo “datetime” para poder ser persistido. Por otro lado, se define el método “to_serial”, que recibiendo un dato de tipo datetime, se encarga de serializarlo.

Además, se puede observar que el campo puede extender el método mágico[4] de Python “__init__”, proveyendo la opción de agregar parámetros de instanciación adicionales a los provistos por el framework. En este caso se podrían indicar parámetros útiles para el formateo en el momento de serializar y des-serializar los datos.

3.10.5 Configuraciones opcionales de los bloques

Además de poder configurar los campos o datos a utilizar y la lógica de la operación de un bloque, también se pueden definir algunos aspectos adicionales.

Como se observa en la figura 60, las configuraciones opcionales de un bloque se encuentran bajo la sub-clase “Meta” que se define dentro de un bloque, e indica toda la configuración del bloque que no tiene que ver con campos o con la lógica de ejecución en sí.

A continuación, se listan todos los atributos que se pueden definir para esta clase, junto con su significado o utilidad y su valor por defecto:

- ✓ **category_name:** es un string que indica el nombre de la categoría en la que el bloque se debe listar. Al alterar este nombre, el bloque aparecerá bajo la categoría indicada cuando se piden los datos de todos los bloques para general los paneles de herramientas visuales. El valor por defecto cuando no se explicita es “Others”.
- ✓ **display_name:** este es el nombre “bello” o para mostrar del bloque definido. Si se omite se toma el nombre de la clase del bloque como su valor por defecto.
- ✓ **disabled:** un booleano que indica si el bloque está habilitado para ser utilizado y listado bajo el panel visual de bloques. Su valor por defecto es “False”.
- ✓ **persist_after_execution:** es un booleano que indica cuándo el estado de un bloque debe ser almacenado luego de ser ejecutado. Esta funcionalidad se explica detalladamente en la sección 3.10.8.
- ✓ **hide_trigger_input:** en algunos casos, para algunos bloques puede ser deseable ocultar el campo de ejecución condicional que se explicó en la sección 3.6. Este atributo es un booleano que indica cuándo debe ser ocultado. Su valor por defecto es “False”**.**

---

[4] Los "magic methdos" también conocidos como "special methods", son métodos especiales que Python provee de manera implícita para agregar comportamiento especial a clases e instancias de clases. Si bien es difícil encontrar una documentación clara y precisa de los mismos, la mayoría de los programadores de Python conocen su existencia y hacen uso de los mismos. Uno de los métodos más utilizados es __init__, que es utilizado para la construcción/inicialización de instancias de una clase [36].

3.10.6 Bloques para notificación/envío de resultados a sistema general

Dado que cada aplicación o sistema general que pueda hacer uso de Flowgramming poseerá su propia interfaz y métodos en los cuales recibir resultados provenientes del framework, no se define ninguna generalidad acerca de cómo ni cuándo el mismo devolverá resultados de los diagramas a la aplicación.

Esta funcionalidad deberá ser implementada a través de un bloque customizado por cada desarrollador que integre el framework. Por ejemplo, se podría definir un bloque que, al recibir una entrada, llame a un método de la aplicación para informar los resultados, de forma que enviará al sistema general los datos bajo una clave:

```
class SendDiagramResultBlock(GenericBlock):
    output_data_type = float

    input = fields.FloatField(disable_default_value=True)
    key = fields.CharField(disable_connector=True)

    class Meta:
        display_name = "Send Diagram Result"

    def execute(self):
        send_data_to_app(self.input, self.key)
        return self.input
```

Figura 72: Bloque de envío de resultados

Si bien podría decirse que esta forma de notificar resultados podría ser un poco tediosa debido a que el desarrollador deberá necesariamente crear un bloque para tal fin, también posee la ventaja de que estos bloques podrían ser insertados en cualquier parte del diagrama para informar resultados en diferentes etapas del proceso, como se muestra a continuación:

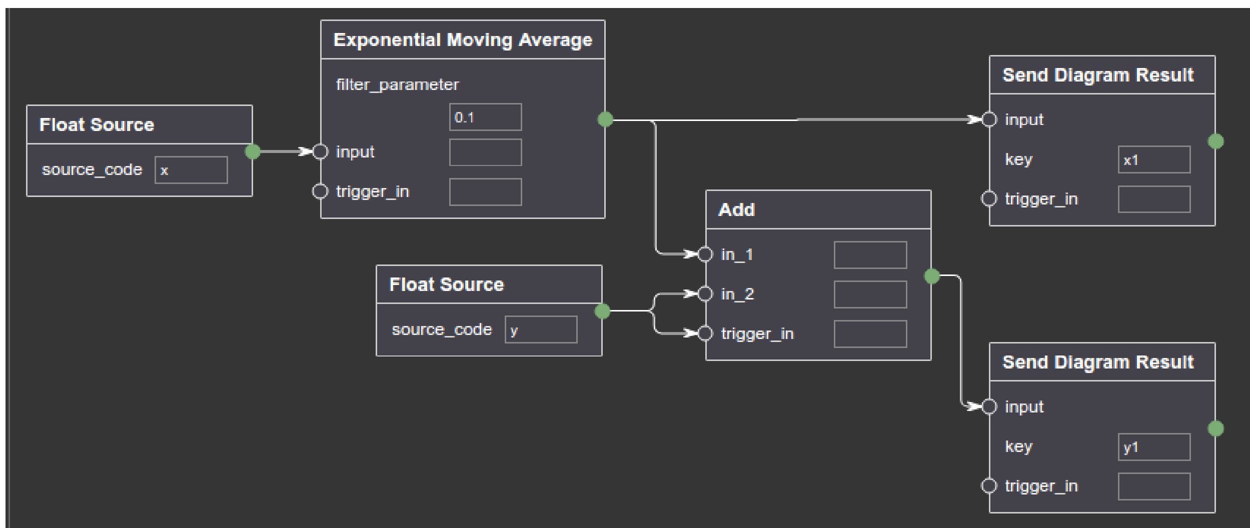

Figura 73: Uso de bloque de envío de resultados en diferentes etapas de un diagrama

3.10.7 Interrupción de ejecución de bloques y diagramas

Cuando un desarrollador define un nuevo bloque, podría necesitar frenar la ejecución del bloque y consecuentemente de la propagación de los resultados por el resto del diagrama para evitar la ocurrencia de una excepción por datos inesperados.

Un ejemplo claro podría ser el de querer evitar la ejecución de un bloque de división si el dato recibido como divisor es igual a 0. Para ello, el framework provee un tipo especial de excepción que este capta y utiliza para parar automáticamente la ejecución del bloque y, consecuentemente, la propagación de datos a través de los diferentes bloques relacionados. Esta clase se denomina "StopExecutionException" y se encuentra bajo el módulo "blocks" de Flowgramming. A continuación, se muestra el ejemplo planteado:

```python
from flowgramming_framework.blocks import StopExecutionException


class SafeDivisionBlock(GenericBlock):
    output_data_type = float

    dividend = fields.FloatField()
    divider = fields.FloatField()

    class Meta:
        display_name = "Safe Division"

    def execute(self):
        if self.divider == 0:
            raise StopExecutionException()
        else:
            return self.dividend / self.divider
```

Figura 74: Bloque de división segura

Notar que cuando se usa este tipo de "parada" de la ejecución, la misma solo afectará al bloque en cuestión y a los bloques que se conectan a la salida de este, mientras que el resto de los bloques del diagrama que se ejecutan en otras ramas no serán afectados y seguirán su ejecución de forma regular.

3.10.8 Memorización de estado

La última funcionalidad de gran importancia presentada en la sección 3.10 es la de persistencia o memorización de estado. Por defecto, los bloques no garantizan el almacenamiento del estado interno de los campos o "fields". Se dice que no se garantiza, porque para la ejecución del framework, en algunas circunstancias se guardan algunos datos, pero esto es algo que el framework decide y que es y debe ser transparente para el desarrollador.

En algunas circunstancias, podría presentarse el caso en el que el desarrollador desee almacenar algún tipo de estado de los campos de un bloque. Por ejemplo, si tuviera que definir un bloque cuya funcionalidad es la de acumular las entradas recibidas, necesitaría que este campo vaya sumando y almacenando los valores luego de cada ejecución.

Para tal fin, se provee el atributo "persist_after_execution" dentro de la clase "Meta" de un bloque. Cuando este atributo posee el valor "True", luego de la ejecución de un bloque se almacena el valor de cada uno de sus parámetros o campos, de modo que estarán disponibles en la próxima ejecución del bloque.

A continuación, se muestra un ejemplo del uso de esta funcionalidad:

```
class AccumulatorBlock(GenericBlock):
    output_data_type = float

    input = fields.FloatField()
    accumulated_value = fields.FloatField(is_input=False)

    class Meta:
        display_name = "Accumulator"
        persist_after_execution = True

    def execute(self):
        if self.accumulated_value is None:
            self.accumulated_value = 0

        self.accumulated_value += self.input

        return self.accumulated_value
```

Figura 75: Ejemplo de uso de persistencia post-ejecución

3.11 Creación de backends de persistencia

Como ya se ha detallado en los capítulos anteriores, los backends de persistencia son los que permiten al framework almacenar y obtener diagramas en y desde algún medio de almacenamiento definido.

Ya se ha explicado en la sección 3.8.2 que el framework provee un backend para almacenar los diagramas en memoria, de forma temporal y volátil, y otro backend para almacenar en disco y tener una especie de "caché" de los diagramas en memoria.

Para desarrolladores con proyectos específicos, es muy probable que ninguno de estos dos backends provistos sea suficiente, y que requiera de alguna forma de almacenamiento específica.

Para cumplir con tal requerimiento, el framework permite que el desarrollador defina sus propios backends de persistencia, extendiendo de una clase base abstracta provista, e implementando sus métodos abstractos o hooks.

La clase abstracta base se encarga de la lógica más pesada, como la serialización de los bloques, las conexiones y los diagramas, la validación de los mismos y otras tantas funcionalidades, delegando en los desarrolladores las operaciones básicas que tienen que ver con el almacenamiento de los objetos ya listos para ser persistidos.

La clase base y abstracta que el framework provee, se denomina "AbstractPersistenceBackend" y se puede encontrar bajo el módulo "persistence" del framework.

A continuación, se listan los métodos básicos y obligatorios para implementar, junto con sus parámetros, y una descripción de lo que deberían hacer:

- ✓ persist_diagram_data(self, _id, name, execution_mode, enabled): dados los parámetros indicados para un diagrama (el identificador, el nombre, el modo de ejecución y si está habilitado o no), el desarrollador debe definir como almacenar estos datos para un diagrama.
- ✓ get_diagram_data(self, diagram_id): dado el identificador de un diagrama, el desarrollador debe retornar un diccionario Python con los datos almacenados por el método previo, siguiendo la siguiente estructura:

```
{
   "id": id del diagrama,
   "name": nombre del diagrama,
   "execution_mode": modo de ejecución del diagrama,
   "enabled":  un booleano indicando si el diagrama está habilitado o no,
}
```

- ✓ delete_diagram_data(self, diagram_id): dado el identificador de un diagrama, este método debe definir cómo eliminar todos los datos del mismo.
- ✓ get_diagrams_ids(self): debe devolver una lista con todos los identificadores de los diagramas almacenados
- ✓ persist_serialized_block(self, block_id, serialized_block, diagram_id, is_source_block): dado el identificador de un bloque, su representación serializada (en forma de String), el identificador del diagrama y si es o no un bloque de tipo "source", este método debe almacenar todos estos atributos para luego poder retornarlos.
- ✓ persist_diagram_connection(self, diagram_id, source_block_id, target_block_id, target_block_input_name): dado el identificador de un diagrama y los atributos de un enlace, compuesto por los identificadores del bloque origen, del bloque objetivo y el nombre del parámetro del bloque objetivo, este método debe almacenar una conexión perteneciente a un diagrama.

- ✓ get_serialized_block(self, diagram_id, block_id): dado el identificador de un diagrama y de un bloque, este método retorna el bloque serializado previamente almacenado, que el framework se encargará luego de des-serializar.
- ✓ get_diagram_block_ids(self, diagram_id): dado el identificador de un diagrama, este método debe retornar todos los identificadores de los bloques que lo componen.
- ✓ get_input_connections_for_block(self, block): dada una instancia de un bloque (de la cual se puede hacer uso los metodos "get_diagram_id" y "get_id"), este método debe retornar un listado con todas las conexiones de entrada para el mismo, donde cada conexión debe ser de la forma:

  (id_del_bloque_origen, nombre_del_parámetro)
- ✓ get_output_connections_for_block(self, block): dada una instancia de un bloque (de la cual se puede hacer uso los métodos "get_diagram_id" y "get_id"), este método debe retornar un listado con todas las conexiones de salida para el mismo, donde cada conexión debe ser de la forma:

  (id_del_bloque_destino, nombre_del_parámetro_del_bloque_destino)
- ✓ delete_block(self, block_id, diagram_id): dados los identificadores de un diagrama y de un bloque, este método debe definir cómo eliminar de forma definitiva el bloque persistido.
- ✓ delete_connection(self, diagram_id, source_block_id, target_block_id, target_block_input_name): dados los atributos de una conexión de un diagrama, este método debe definir cómo eliminar de forma definitiva la conexión.

Además de estos métodos, el desarrollador podría extender el método mágico __init__ de Python de la clase del backend para recibir parámetros posicionales de configuración cuando el framework es instanciado como ya se ha explicado. Por ejemplo, si un desarrollador quisiera almacenar los bloques en una base de datos, podría definir entre los parámetros, el hostname, usuario, contraseña, nombre de la base de datos, puerto a utilizar, etc.

Los métodos listados son aquellos que son pura y exclusivamente abstractos, y por lo tanto necesarios que el programador desarrolle. Además, existen otros métodos que, si bien no son abstractos y están implementados, podría ser recomendable que el usuario implemente por razones de eficiencia. Debido a que este tipo de personalización está orientada a desarrolladores avanzados, no se provee en este documento información relativa a los métodos. Estos se encuentran marcados y claramente documentados dentro de la clase abstracta y su implementación puede servir de gran ayuda como guía referencial para la implementación de los mismos en las sub-clases.

3.12 Creación de backends de debugging

Cuando un usuario se encuentra desarrollando un diagrama, frecuentemente desea enviar algunos datos de prueba para ver si el diagrama responde y procesa los datos de la forma esperada.

Para que la salida de los bloques pueda ser publicada y leída o notificada en la interfaz visual, el framework hace uso de los backends de debugging o backends de testing.

El framework provee algunos backends básicos, pero permite también la creación de nuevos. La razón por la cual no se proveen backends específicos, como ya se ha expresado, es porque cada sistema podría requerir de diferentes formas de subscribirse a la lectura de salidas de bloques y además porque cada implementación podría requerir de la instalación de diferentes librerías y no tiene sentido requerir la instalación de las mismas para todos los proyectos puesto que la mayoría podrían no utilizarlas.

De todas formas, la creación de un nuevo backend de testing es muy sencilla. Solo se debe crear una clase que extienda de la clase base AbstractOutputValuesDebuggerBackend encontrada en el módulo “debugging” del paquete del framework, y debe implementar un único método denominado “notify_block_output” que recibe un identificador de un diagrama, un identificador de un bloque, y el valor de la salida de ese bloque para ese diagrama.

A continuación, se muestra un ejemplo de la implementación de un backend de testing que envía las salidas de los bloques a través del protocolo MQTT[5] haciendo uso de la librería “paho” desarrollada por el equipo de Eclipse [38]. La parte visual del framework luego se conectará al servicio de MQTT, suscribiéndose y recibiendo los valores notificados, permitiendo notificar y mostrar los valores en la interfaz gráfica.

[5] MQTT es un protocolo de conectividad máquina-a-maquina (M2M) o de internet de las cosas. Fue diseñado como mecanismo de pasaje de mensajes extremadamente liviano [37].

```python
import json
import paho.mqtt.publish as publish
from flowgramming_framework.debugging import AbstractOutputValuesDebuggerBackend


class MQTTOutputValuesDebuggerBackend(AbstractOutputValuesDebuggerBackend):

    def __init__(self, hostname="localhost", port=1883):
        self.hostname = hostname
        self.port = port

    def notify_block_output(self, diagram_id, block_id, value):
        payload = json.dumps({
            'diagram_id': diagram_id,
            'block_id': block_id,
            'value': value
        })

        publish.single(
            "flowgramming_framework_outputs_debugger",
            hostname=self.hostname,
            port=self.port,
            payload=payload
        )
```

Figura 76: Implementación de un backend de Debugging

3.13 Creación de backends de logging de errores

Cuando se integra y ejecuta Flowgramming en entornos de desarrollo, al igual que cualquier otro servicio o aplicación, los errores en tiempo de ejecución pueden ser mostrados en consola de modo que el usuario puede ver qué es lo que falla (si es que así fuera). Pero cuando los errores ocurren en tiempo de ejecución en entornos de producción, es necesario que las fallas ocurridas sean registradas de alguna forma, debido a que no se puede asignar a una persona para que esté constantemente viendo una consola para ver si ocurre un error, y de qué tipo es este.

Como se explicó en la sección de configuración, Flowgramming hace uso de backends para notificar errores. Por defecto notifica los errores en consola, y además provee un backend para persistir los errores en archivos de texto.

Además, Flowgramming también provee las herramientas e interfaces para crear nuevos modos de almacenar o enviar errores de diversas formas. Al igual que con la implementación de nuevos backends de debugging, éstos se pueden crear simplemente extiendo de una clase base denominada “AbstractErrorLoggingBackend” ubicada en el módulo “errors_logging” del paquete de Flowgramming, y el desarrollador sólo debe implementar un método denominado “log_exception_message” el cual recibe el “stacktrace” de la excepción ocurrida y se debe definir qué hacer con este mensaje.

A continuación, se presenta un ejemplo de implementación de un backend en el que se envían por mail los mensajes de excepciones ocurridas:

```
from flowgramming_framework.errors_logging import AbstractErrorLoggingBackend


class EmailErrorLoggingBackend(AbstractErrorLoggingBackend):
    def __init__(self, emails_list):
        super(EmailErrorLoggingBackend, self).__init__()
        self.emails_list = emails_list

    def log_exception_message(self, error_message_formatted):
        recipients = self.emails_list
        subject = "Flowgramming Exception"
        content = error_message_formatted
        send_email(recipients, subject, content)
```

Figura 77: Implementación de un backend de notificación de errores

Para el caso propuesto, al instanciar el framework se deberá pasar entre los parámetros de configuración una lista con los mails a los que se desea enviar la notificación de errores.

Capítulo 4: Notas de implementación de Flowgramming Framework

4.1 Introducción

Este breve capítulo tiene como objetivo mencionar y detallar algunas técnicas de programación utilizadas para llevar a cabo la implementación de algunas características particulares del framework, así también como la comparación frente a otras técnicas.

4.2 Variables de Instancia Declarativas

Cuando se detalló cómo definir un tipo de bloque, se pudo observar que, dentro de la declaración de la clase del mismo, la declaración de un campo que luego es utilizado durante el procesamiento, se realiza de la siguiente forma:

```
class MyBlock(GenericBlock):
    field_1 = fields.FloatField()
    field_2 = fields.CharField(is_input=False)
```

Figura 78: Clase Declarativa

Este tipo de declaración de variables o atributos de instancia a través de variables de clase es un común patrón de diseño encontrado en varias herramientas, el cual, a priori, parecería no tener un nombre en particular.

Generalmente, se hace referencia a este patrón de diseño con diferentes términos, tales como "Declarative Style Class API", "Declarative Class Definitions", "Declarative API Classes" o "Declarative Class Attributes". De nuevo, la característica más relevante de este patrón o técnica es el uso de variables de clase para modelar variables de instancia bajo el mismo nombre.

Hoy en día este patrón de diseño ha tomado una gran relevancia en una amplia variedad de herramientas desarrolladas en Python, generalmente usados para transformación de modelos y mapeos de datos, aunque no está limitado a ningún fin en particular, siendo útil para la implementación de diversas soluciones.

Este patrón brinda al menos tres ventajas relevantes:

1. Python no es un lenguaje en el cual las variables de instancia se declaren como parte de la clase, sino que, en cambio, las variables de instancia son creadas o asignadas en el método de inicialización __init__. Por lo tanto, este patrón brinda la posibilidad de declarar variables de instancia incluso cuando Python no provee ese comportamiento por defecto.
2. Por otro lado, permite agregar o definir características o configuraciones propias para cada atributo, que luego pueden ser utilizadas en tiempo de ejecución para definir

diferentes aspectos de los mismos. Esto es claramente visto en el framework en la declaración de campos, donde como se mencionó en la sección 3.10.4.2, cada una de estas opciones definen meta-datos utilizados para alterar la forma en la que los atributos de un bloque son mostrados o se comportan.

3. Permite analizar las instancias creadas junto con sus atributos para que estas puedan ser mapeadas a otro tipo de objetos o serializadas.

Este patrón, visto en la implementación de muchas herramientas y frameworks, como por ejemplo los ORM SQL Alchemy[39] y el ORM de Python/DjangoDjango[40], y utilizado en múltiples herramientas que hacen uso de declaración de modelos como Scrapy[41], Haystack[42] y muchas más, resulta bastante intuitivo para muchos desarrolladores y generalmente no necesita de explicación previa a su uso. Generalmente en la documentación, se muestra directamente cómo es utilizado, sin detallar explícitamente como es que funcionan, dado que su uso resulta bastante natural. Por estos motivos se ha elegido como el método para la declaración de nuevos bloques en el framework.

Para llevar a cabo la implementación de este tipo de clases declarativas, se utilizó como base el código de la herramienta Python Micromodels[43], pero se le han realizado una serie de modificaciones para que encaje con el framework y sus objetivos, eliminando funcionalidades no requeridas y agregando faltantes.

Otro aspecto relevante que se relaciona con la declaración de clases es la de las variables de clase "Meta" pertenecientes a cada clase de bloque, que representan una sub-clase interna que define todos los meta-datos relativos a la clase, que no son campos (fields). Es decir, provee información adicional que permite agregar información adicional de un bloque. Este esquema fue tomado de la declaración de modelos de Python/Django.

La combinación de las técnicas de clases declarativas, junto con la declaración de meta-datos, provee una potente forma de declarar y customizar clases de una forma clara, flexible y sencilla.

Además, esta combinación de técnicas brindará todo lo necesario para crear mejores alternativas de serialización de bloques, tema que se mencionará en el capítulo de trabajos futuros.

4.3 Localización de clases

El framework provee un sistema de localización de clases utilizado para encontrar, principalmente, las clases relativas a los siguientes elementos:

- ✓ **Backends**: para encontrar backends de debugging, de notificación de errores y de persistencia
- ✓ **Bloques definidos por el desarrollado**r: para encontrar nuevos tipos de bloques definidos por el usuario.

Al instanciar una aplicación del framework, se pudo observar que se indica, de forma relativa, las direcciones de cada una de las clases de los backends, y de los módulos de Python contenedores de bloques customizados o definidos por el desarrollador:

```
flowgramming_app = FlowgrammingApp(
    'CachedFileHashedPersistenceBackend',
    ["/var/flowgramming/persistence_file.txt"],
    custom_blocks_paths=[
        'web_app.flowgramming.custom_blocks'
    ]
)
```

Figura 79: Ejemplo de especificación de rutas relativas para bloques y backends

Para el caso de los backends, la forma de definir qué backend utilizar varía de acuerdo a si el backend es uno provisto por el framework o uno que el usuario define. En el primer caso, con especificar el nombre de la clase del backend es suficiente y el framework sabrá donde buscar la clase para cada tipo de backend en particular. Cuando en cambio se selecciona un backend definido por el usuario, como ya se ha especificado, la manera de indicarlo es mediante una ruta o camino relativo al módulo que define la clase del backend.

La razón por la cual se eligió esta forma, es porque brinda una forma de seleccionar las diferentes clases de manera universal a través de strings, tanto para clases que el framework provee como para clases que el usuario define, sin la necesidad de importar los módulos o clases al momento de instanciar una aplicación del framework. Si sólo se indicaran clases definidas por el usuario, se podrían importar y referenciar de forma directa, y este mecanismo no sería de mayor relevancia ya que el usuario debería saber la localización de sus clases. Pero resulta incómodo cuando se hace uso de clases provistas por el framework, debido a que el usuario debería conocer las localizaciones de cada clase.

Además, existen dos motivos adicionales por las cuales se decidió utilizar este método. El primero es porque es comúnmente utilizado por varias herramientas, por lo que resultará de familiaridad para el desarrollador. Y el segundo es porque si en posteriores versiones del framework la localización de las clases se ve alterada, el usuario no necesitaría actualizar sus configuraciones.

Para el caso de los bloques definidos por el desarrollador es similar, basta con especificar la ruta relativa del módulo o paquete que contiene las clases de los bloques definidos (la ruta relativa al proyecto que hace uso de Flowgramming), y el framework sabrá cómo encontrar todos los bloques que extienden de la clase base "GenericBlock", para proveer estos bloques de forma automática tanto para la generación de los paneles como para la ejecución de los mismos.

4.4 Ejecución de diagramas

El framework para la ejecución de un diagrama desde que llega un dato para una clave hasta que termina la ejecución del mismo, ejecuta una serie de pasos, entre los que se incluye la identificación de los diagramas relacionados con la clave enviada, el envío de los datos a los bloques pertinentes y la ejecución de los bloques relacionados con los bloques iniciales relacionados a la clave enviada.

Más detalladamente, la serie de pasos al recibir un valor para una clave es:

1. Buscar, para todos los diagramas, los bloques de tipo “source” (es decir, los bloques que permiten la entrada de datos a un diagrama) que se relacionan con la clave enviada
2. Para cada uno de los bloques, de forma iterativa, se envía el dato recibido por el framework
3. Cada uno de los bloques que recibe un dato entrante y está habilitado para su ejecución, ejecuta su operación y luego, de forma recursiva, reenvía el resultado de su ejecución a todos los bloques que debe reenviar su salida, definido por sus enlaces. Cada uno de estos bloques ejecuta este mismo paso en su interior, hasta que la ejecución se termina porque:
    a. Se ha llegado al extremo de un diagrama, donde ya no hay más enlaces por los cuales enviar datos
    b. Alguno de los bloques no posee todos los parámetros necesarios para ejecutar su operación,
4. El proceso finaliza y el framework queda a la espera de la recepción de datos para comenzar por el paso 1 nuevamente.

Como se puede observar, en el paso número 3, se realiza una ejecución recursiva para hallar los bloques adyacentes (directamente conectados) a otro bloque y reenviar los datos luego de que cada bloque ejecuta su operación.

Al momento de la implementación, se implementó y se evaluó también un enfoque iterativo en vez de la utilización del recursivo, con el fin de utilizar una menor cantidad de recursos en tiempo de ejecución. Sin embargo, frente a la ocurrencia de ciclos infinitos, el enfoque recursivo presenta una ventaja, ya que la ejecución alcanza un límite de recursión máximo, lanzando una excepción especial y permitiendo frenar la ejecución y notificar (mediante las técnicas de notificación de errores) el suceso. Con el enfoque iterativo, la identificación de este evento sería algo más compleja y requeriría de estructuras de datos adicionales.

Si bien no existe mucha documentación acerca de cómo las herramientas y lenguajes de programación de flujo de datos implementan la ejecución de sus flujos, algunas especifican, a grandes rasgos, que poseen una especie de tabla que se poseen todos los bloques y el estado de sus entradas para la evaluación de su ejecución. Este método podría ser una notable mejora en términos de escalabilidad y performance, pero requeriría de un gran esfuerzo para la contemplación de varios casos especiales. Es por eso que se presenta en la sección 7.4 en el capítulo de trabajos a futuro realizables sobre el framework.

4.5 Validación de diagramas

La validación de diagramas previo a ser persistidos no posee una lógica demasiado compleja. Sin embargo, posee algunos detalles relevantes para ser mencionados.

Como primer paso, se verifica que cada uno de los bloques definidos de forma aislada poseen todos los datos y parámetros requeridos, verificando que los mismos puedan ser instanciados y ejecutados en tiempo de ejecución.

Luego se verifica para todos los enlaces definidos, que estos sean válidos. Entre la lista de condiciones a verificar se encuentran:

- ✓ Que los identificadores de los bloques a cada extremo del enlace sean válidos y que se correspondan con bloques definidos para el diagrama
- ✓ Que el nombre del parámetro del bloque objetivo al que se enlaza la salida del bloque origen sea correcto
- ✓ Que el tipo de dato de la salida del bloque origen y el tipo de datos aceptado por el parámetro del bloque destino sean compatibles
- ✓ Que el parámetro del bloque origen tiene, a lo sumo, un solo enlace conectado

Una vez que se han validado los bloques y sus enlaces, se procede a la verificación de presencia de ciclos en los diagramas. Básicamente, para llevar a cabo esta acción, primero se representa al diagrama como un grafo, donde cada bloque es un nodo y cada enlace una arista. Posteriormente se evalúan y se listan los posibles ciclos presentes a través de un recorrido de grafos en profundidad (DFS, Depth First Search) con verificación de ciclos. Una vez que se poseen todos los ciclos listados, se procede a la evaluación de los mismos, analizando los bloques componentes de los ciclos junto con el modo en el que se desea ejecutar el diagrama (memorable o consumidor), para clasificar los ciclos en críticos y no críticos, bajo las reglas ya explicadas en la sección 3.7.1.

Capítulo 5: Casos de uso / Casos de ejemplo

5.1 Introducción

En este capítulo se desarrollarán de forma breve algunos casos de uso en los que Flowgramming puede ser utilizado para resolver problemas específicos.

Se comenzará con la solución de los problemas o casos planteados en la propuesta del trabajo, donde se concluirá en las soluciones sin la necesidad de hacer uso de bloques adicionales a los provistos por el framework. Luego se continuará con el desarrollo de algunos casos adicionales en los que se mostrará cómo Flowgramming puede contribuir a la solución de diversos problemas, desarrollando nuevos bloques de forma simple y rápida.

Para el desarrollo de algunos de los ejemplos, se implementó una funcionalidad que permite la visualización de un gráfico en tiempo real que permite seguir a los valores procesados y sin procesar que son enviados al framework. Esta funcionalidad se mostrará en las capturas de pantalla incluidas para cada caso. Además, para mostrar algunos de los ejemplos, se desarrollaron pequeños métodos que envían datos de prueba al framework de forma automática y aleatoria.

5.2 Ejemplo de la propuesta #1: Calibración de un sensor de sonido

El primer caso presentado en la propuesta inicial se trata de un problema sencillo que permite mostrar de forma simple cómo el framework puede resolver un problema. La descripción del problema es la siguiente:

> En el centro de una ciudad se requiere medir y analizar datos sobre el ruido de la misma para hallar si se encuentran niveles críticos que alteran el bienestar de las personas. Además, la entidad que debe desarrollar la solución no dispone de muchos recursos, por lo que utilizará sensores económicos, no programables, y donde cada uno posee un leve error o desviación que varía entre los diferentes sensores. Se necesita que cuando cada empleado vaya colocando los sensores en distintos puntos de la ciudad, pueda ir calibrándolos mediante un servicio web comparando los resultados con los de un artefacto de medición de sonido de alta precisión.

Con el fin de resolver un ejemplo sencillo dentro de este caso, se supone que en el primer sensor que instala el empleado se detecta una diferencia negativa de 5 decibeles cuando se compara el valor del sensor con el del artefacto de alta precisión. Esta diferencia podría rápidamente ser corregida en el servicio web generando el siguiente esquema, que adiciona los 5 decibeles nuevamente:

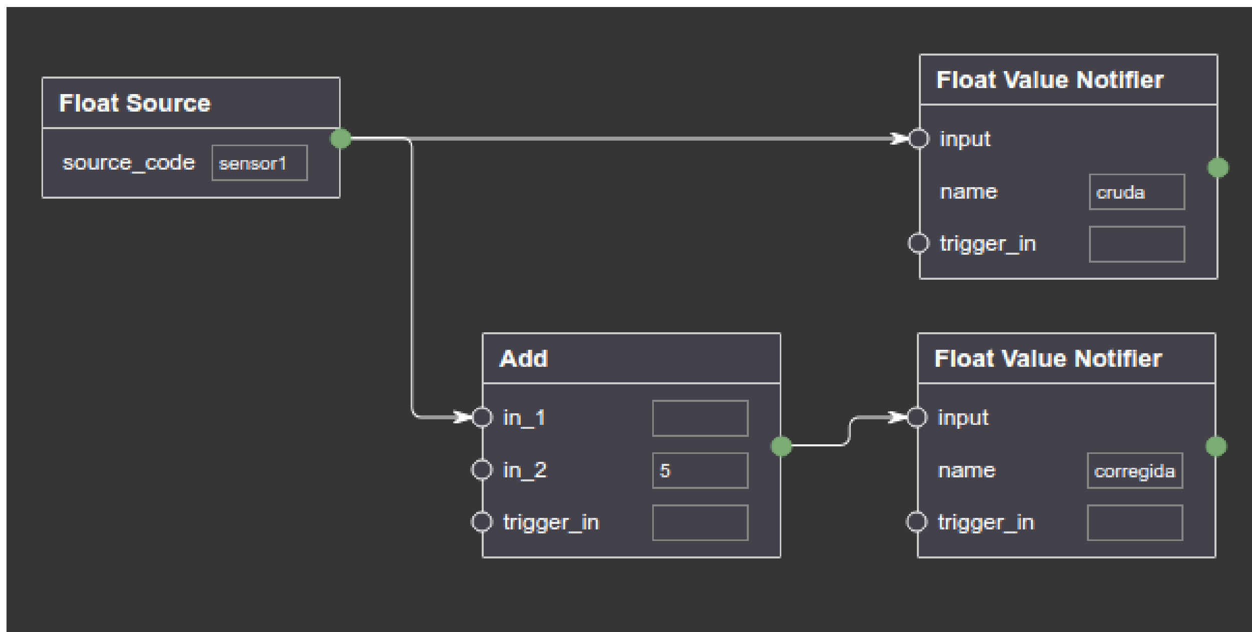

Figura 80: Diagrama del caso de ejemplo #1

Los dos bloques ubicados en el extremo derecho de la figura (de tipo Float Value Notifier), son bloques personalizados creados a los efectos de mostrar la salida procesada (corregida) y sin procesar (cruda) del sensor de sonido en un gráfico de tiempo real como el siguiente:

Figura 81: Salida de la solución del caso de ejemplo #1

Al ir colocando otros sensores, el empleado puede ir corrigiendo la salida del sensor sumando, restando o aplicando otras funciones a la salida producida por cada sensor.

5.3 Ejemplo de la propuesta #2: Análisis de Sonido en ciudad eliminando ruidos

El siguiente caso de ejemplo presentado en la propuesta está en el mismo dominio del ejemplo anterior, donde luego de las primeras ejecuciones y análisis de los primeros datos en un sensor X, se halló que en ciertos momentos se generan picos aislados que distorsionan el análisis de los datos. Por lo tanto, se desea encontrar una solución que permita eliminar o disminuir estos picos para obtener datos más representativos.

Se resolverá el problema de dos formas posibles: una en la que se eliminan los picos presentes (solución "A"), y otra en la que se disminuyen o suavizan los mismos (solución "B"). La idea es mostrar que hay varias soluciones, mientras que la decisión final radicará en el caso de uso específico de la vida real, dependiendo de cuál sea exactamente el resultado buscado.

Solución A: Para esta solución, se evitará enviar la salida del sensor, si la diferencia entre dos valores recibidos consecutivos es mayor que un número "Y" (para este ejemplo, se utiliza arbitrariamente el número 15):

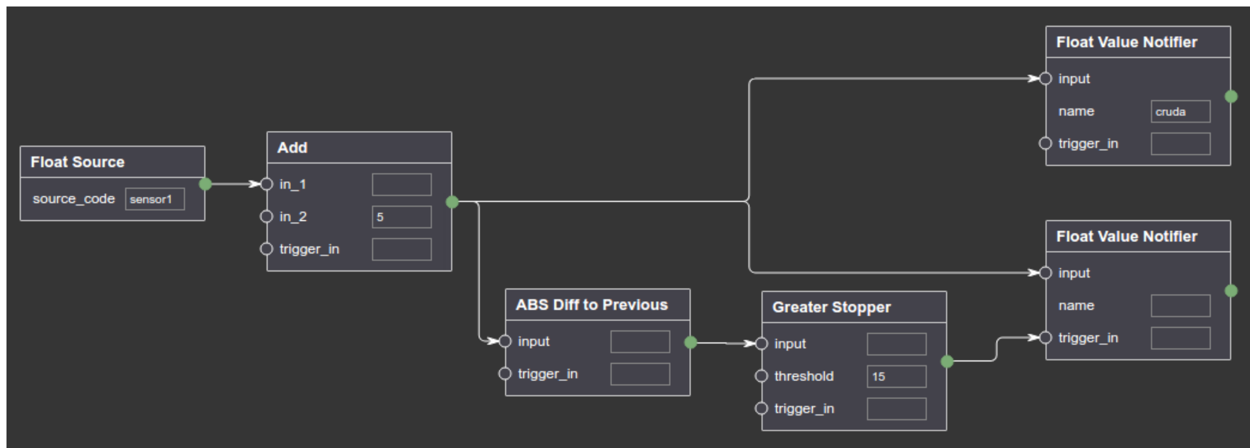

Figura 82: Diagrama del caso de ejemplo #2 A

Como se puede observar, la corrección del error sigue estando, ya que se le sigue sumando 5 a la salida del sensor. Luego, se calcula la diferencia entre los valores consecutivos mediante el bloque "ABS diff to previous" (diferencia absoluta contra el valor previo) y luego un bloque de tipo "Greater Stopper", que frena el envío de datos si la diferencia recibida es mayor a 15db.

Como resultado, se obtiene una salida similar a la siguiente, en la que se comparan los valores sin ser tratados y luego de ser tratados para eliminar picos:

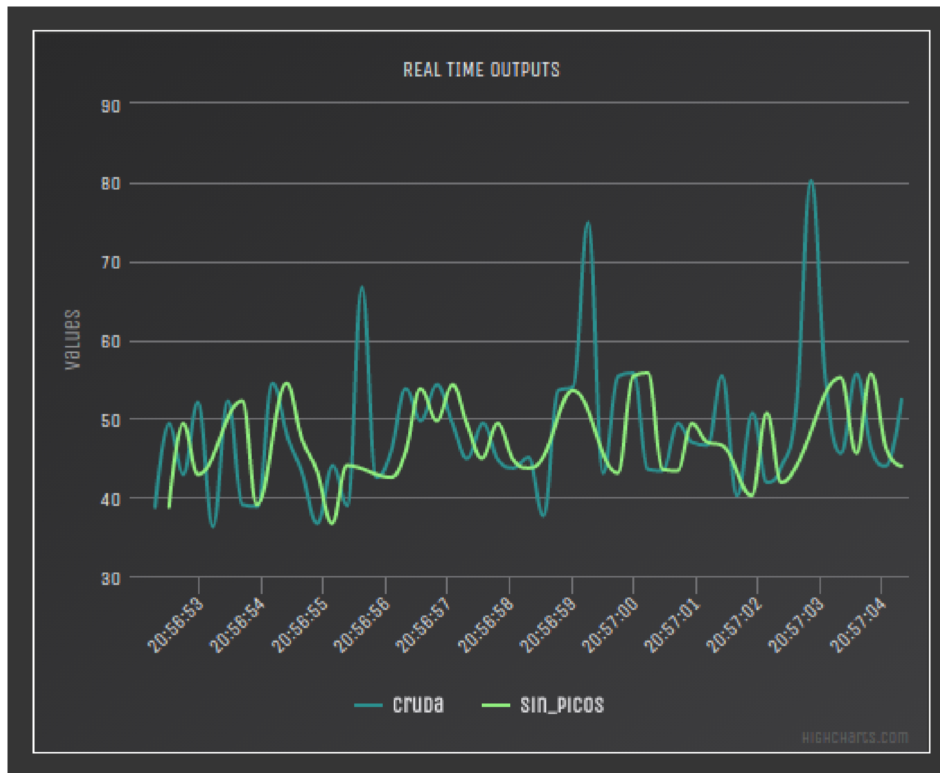

Figura 83: Salida de la solución del caso de ejemplo #2 A

Puede verse en la salida procesada para evitar los picos (en un verde más claro), que los picos enviados por el sensor no se encuentran presentes, fueron eliminados.

Solución B: Esta segunda solución, en contraste con la anterior, suaviza los picos (y en general, un poco la diferencia entre todos los valores consecutivos) en vez de eliminar picos, aplicando la media móvil exponencial la salida del sensor:

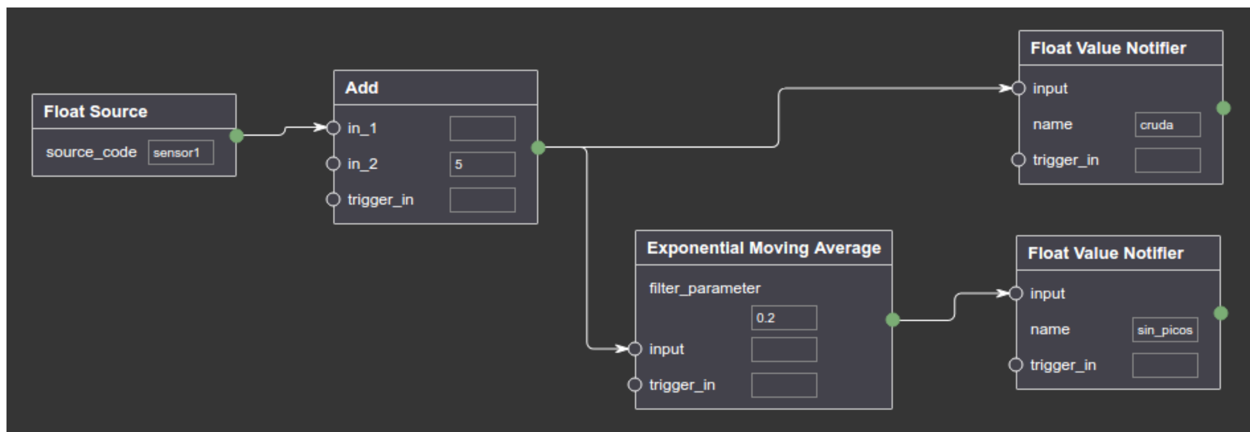

Figura 84: Diagrama del caso de ejemplo #2 B

Como se puede observar en el diagrama, en el bloque "Exponential Moving Average" se aplica la media móvil exponencial, en la que cada nuevo valor tiene un peso menor (en este caso del 20%) sobre el conjunto anterior, generando una salida que reduce los picos registrados en la entrada, y las diferencias generales entre valores subsecuentes o adyacentes:

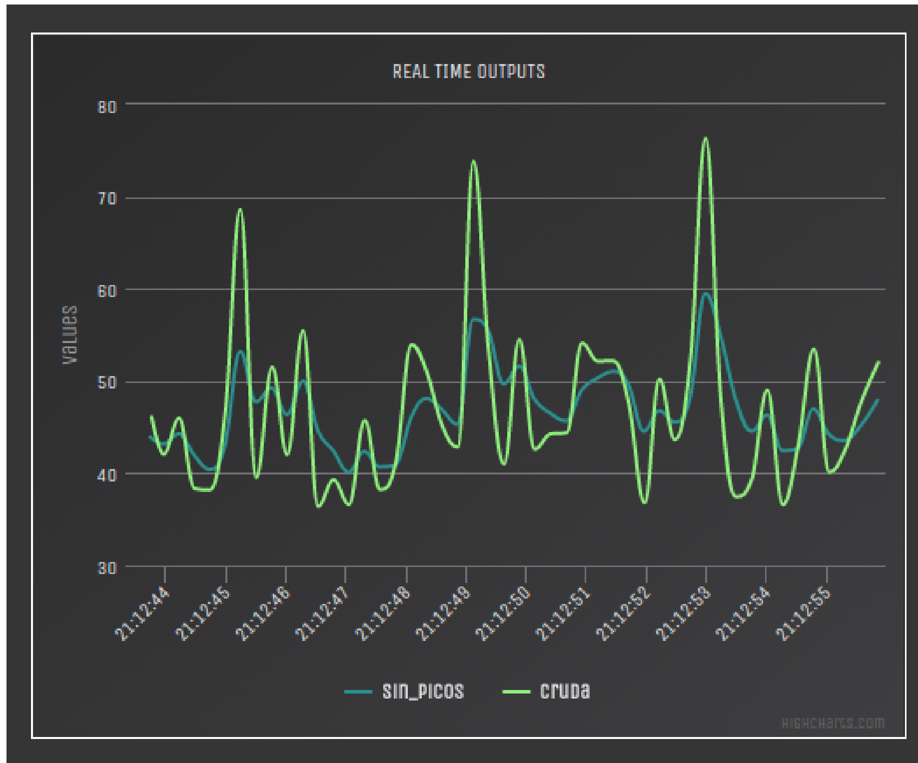

Figura 85: Salida de la solución del caso de ejemplo #2 B

5.4 Ejemplo de la propuesta #3: Detección de fallas en maquinaria según vibraciones.

El último caso presentado en la propuesta, presenta el siguiente problema:

> "En una fábrica de acero se poseen máquinas de producción de alto rendimiento. Las mismas, dependiendo de la etapa de producción en la que se encuentren, podrían presentar vibraciones "normales" en uno de sus dos extremos, pero si la vibración pasa un cierto límite en ambos extremos, esto podría indicar la presencia de un problema, por lo que se desea notificar a un supervisor para que la revise."

A continuación, se muestra un posible diagrama que podría resolver el problema. El mismo tiene como salida un valor booleano, que indica cuándo se debe lanzar una notificación de error en una máquina. Para mostrar el diagrama con el mejor detalle posible, se eliminaron los bloques de notificación:

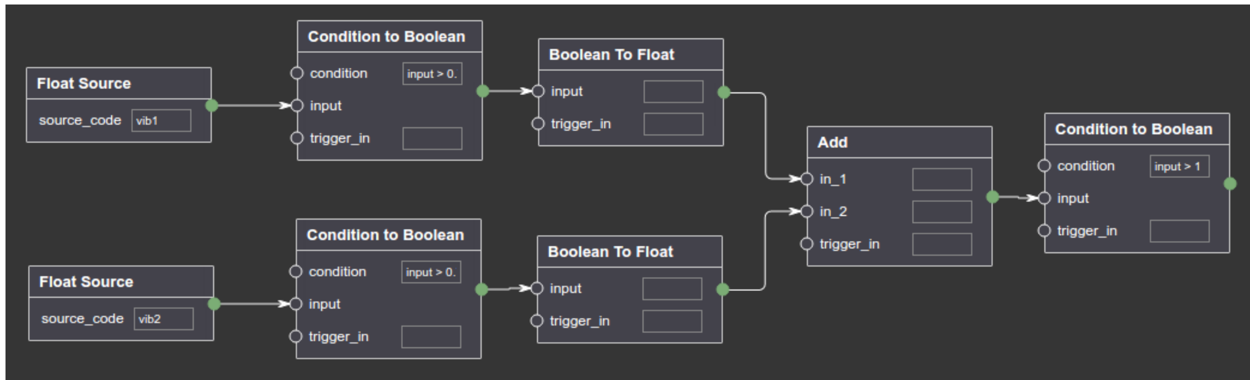

Figura 86: Diagrama del caso de ejemplo #3

El primer bloque de cada rama (donde cada rama representa a cada extremo de la máquina), evalúa si la vibración sensada es mayor que un cierto límite, lo que mostraría que la vibración es alta. Luego, el resultado de esa evaluación se convierte en un dato de tipo numérico, representando el '1' una alta vibración y el '0' una vibración regular o baja. Posteriormente se suman los resultados provenientes de ambas ramas y se evalúa si esta suma es mayor que 1, lo que indicaría una falla en la máquina, en concordancia con el enunciado del problema.

A continuación, se puede observar en el gráfico de tiempo real la salida del diagrama anterior, donde las líneas azules y rojas indican las vibraciones a cada extremo de la máquina, y la línea verde indica si en un instante dado se da un caso de falla:

Figura 87: Salida de la solución del caso de ejemplo #3

5.5 Manejo seguro de una cinta industrial o de un sistema de aerosillas

Con el fin de demostrar que se pueden resolver casos complejos a través de los bloques base que provee Flowgramming, se presenta el siguiente ejemplo:

> "Se desea modelar un sistema de una cinta de producción industrial, en la cual se poseen dos botones en cada extremo: uno para hacer que la cinta comience a funcionar, y uno para hacer que la cinta se detenga. Estos botones son operados por los operarios de la fábrica, y la cinta debe funcionar de la siguiente forma: para que la cinta funcione, los botones de "comienzo" deben ser oprimidos en ambos extremos de la cinta. Si durante la ejecución se presiona uno de los botones de "parada", luego ambos botones de "comienzo" deben ser presionados nuevamente en cada extremo de la cinta para que la misma comience a funcionar nuevamente."

Este modelo de trabajo, en el que por razones de seguridad ambos extremos deben corroborar que todo está en orden para comenzar la ejecución, también puede ser observado en los mecanismos de aerosillas en los centros de esquí.

El siguiente esquema desarrollado con Flowgramming, cubre el caso planteado:

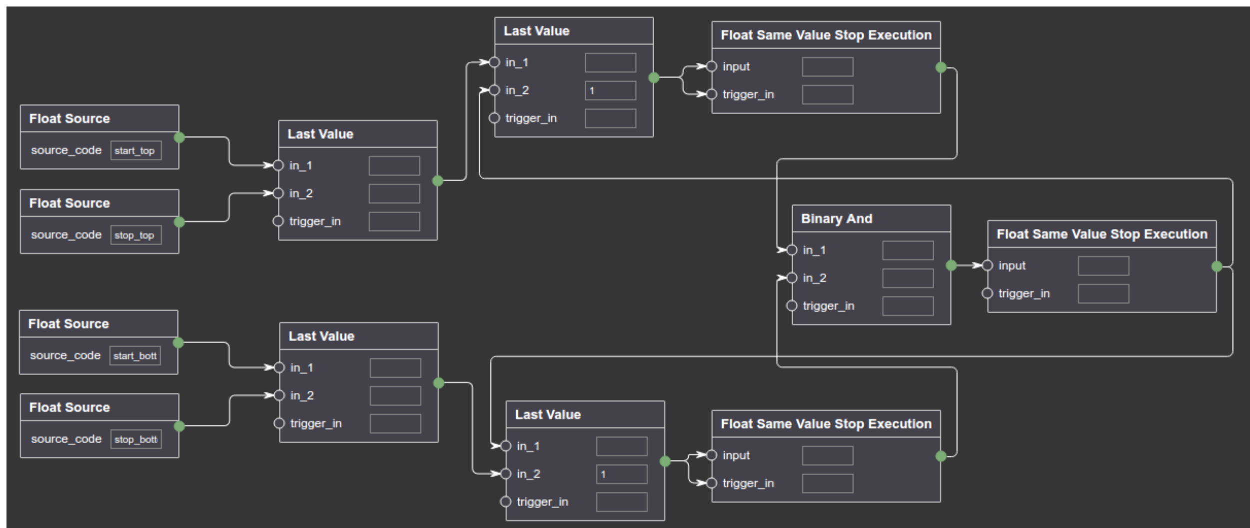

Figura 88: Diagrama de cinta de producción industrial

El diagrama se encuentra ramificado en dos secciones principales, cada una representando un extremo del mecanismo. Luego, cada extremo puede recibir valores de cada uno de los botones (de comienzo y de parada).

Cada rama, envía el último valor recibido por alguno de sus dos botones, a los que luego se les realiza una operación lógica AND para evaluar si la cinta debe funcionar o no. La salida final retroalimenta a las entradas de modo que, si se para la ejecución, se debe volver a enviar valores provenientes de ambos botones de comienzo para la cinta vuelva a funcionar. Además, se

pueden observar bloques que frenan la ejecución si se reciben consecutivamente dos datos con el mismo valor, a modo de evitar ciclos.

Como se ha expresado al comienzo de la sección, la idea de este ejemplo es sólo la de demostrar que se pueden resolver casos complejos y que incluso conservan estado sólo usando bloques que el framework provee. Por supuesto que toda esta lógica podría ser encapsulada en un solo bloque, quedando el diagrama mucho más sencillo.

5.6 Identificación y conteo de objetos en imágenes

Aquí se presenta otro caso en el que se puede observar cómo, con la ayuda de herramientas adicionales, es muy sencillo agregar una nueva funcionalidad que de otra forma podría resultar más compleja de agregar a un sistema.

El caso trata del reconocimiento de objetos en una imagen, en particular en este ejemplo, el reconocimiento de caras. No se entrará en detalle en el tema ni en los diferentes parámetros de configuración de las herramientas utilizas, sino simplemente se mostrará cómo esta característica puede ser fácilmente adicionada.

Se ha desarrollado un nuevo bloque que hace uso de la funcionalidad de reconocimiento de objetos provista por la herramienta de análisis visual OpenCV[46] (referenciada en Python como "cv2"), que hace uso de un archivo que contiene datos de un modelo de entrenamiento acerca de los aspectos generales de objetos (en este caso, de las caras), que permite el reconocimiento de las mismas a través de técnicas de Cascadas Haar[47]. El nuevo bloque posee el siguiente aspecto:

```
class FacesCounter(GenericBlock):
    output_data_type = float
    image_path = fields.CharField(is_input=True)

    class Meta:
        display_name = "Faces Counter"
        category_name = "Image Analysis"

    def execute(self):
        scale_factor = 1.3
        min_neighbors = 4
        this_file_folder = "/".join(os.path.realpath(__file__).split("/")[:-1])
        classifier_path = os.path.join(this_file_folder, "haarcascade_frontalface.xml")

        cv2_image = cv2.imread(self.image_path)
        fac_cascade = cv2.CascadeClassifier(classifier_path)
        gray = cv2.cvtColor(cv2_image, cv2.COLOR_BGR2GRAY)

        res = fac_cascade.detectMultiScale(gray, scale_factor, min_neighbors)
        return len(res)
```

Figura 89: Bloque de conteo de caras

Y es utilizado en los diagramas de la siguiente forma:

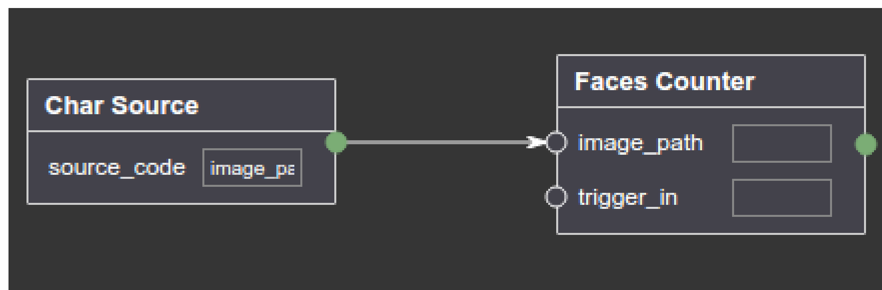

Figura 90: Diagrama de conteo de caras en imagen

Básicamente, el sistema principal almacenará la imagen en un directorio en particular, y enviará la dirección de este directorio a Flowgramming bajo una clave en particular. El nuevo bloque leerá la imagen de este directorio, y como resultado el mismo expulsará un valor numérico que indica la cantidad de caras que ha identificado.

Este tipo de diagramas podría ser utilizado para obtener datos estadísticos. Por ejemplo, en algunas tiendas, se desea ver el movimiento y analizar la cantidad de personas que visitan las mismas en diferentes horarios del día. El sistema podría, periódicamente, ir tomando "frames" de un flujo de video de una cámara IP e ir enviándolos a Flowgramming para analizar la cantidad de personas que se encuentran en la tienda en un momento dado. Quizás para este caso sería útil reemplazar el reconocimiento facial por el reconocimiento de personas o "caminantes", dependiendo de la posición y orientación física de la cámara.

5.7 Identificación y resaltado de objetos en imágenes

Este ejemplo es solo una modificación aplicada al caso anterior. Utiliza las mismas herramientas y la idea es que además de contar los objetos (caras), se genere una copia de la imagen en donde se resalten los mismos.

El código del bloque es el siguiente:

```
class FacesDelineator(GenericBlock):
    output_data_type = float
    image_path = fields.CharField(is_input=True)

    class Meta:
        display_name = "Faces Delineator"
        category_name = "Image Analysis"

    def execute(self):
        scale_factor = 1.3
        min_neighbors = 4
        this_file_folder = "/".join(os.path.realpath(__file__).split("/")[:-1])
        classifier_path = os.path.join(this_file_folder, "haarcascade_frontalface.xml")
        image_path = self.image_path
        image_folder = "/".join(image_path.split("/")[:-1])
        image_name = image_path.split("/")[-1]

        cv2_image = cv2.imread(self.image_path)
        fac_cascade = cv2.CascadeClassifier(classifier_path)
        gray = cv2.cvtColor(cv2_image, cv2.COLOR_BGR2GRAY)

        faces_regions = fac_cascade.detectMultiScale(gray, scale_factor, min_neighbors)

        if len(faces_regions) > 0:
            image_to_draw = Image.open(self.image_path)
            image_draw = ImageDraw.Draw(image_to_draw)
            for rect in faces_regions:
                rect_top_x = rect[0]
                rect_top_y = rect[1]
                rect_bottom_x = rect_top_x + rect[2]
                rect_bottom_y = rect_top_y + rect[3]

                image_draw.rectangle(((rect_top_x, rect_top_y), (rect_bottom_x, rect_bottom_y)), outline=128)
                # more rectangles so the line is stronger:
                image_draw.rectangle(((rect_top_x + 1, rect_top_y + 1), (rect_bottom_x - 1, rect_bottom_y - 1)), outline=128)
                image_draw.rectangle(((rect_top_x + 2, rect_top_y + 2), (rect_bottom_x - 2, rect_bottom_y - 2)), outline=128)

            store_image_path = image_folder + "/" + "faces_" + image_name
            image_to_draw.save(store_image_path, format='JPEG')

        return len(faces_regions)
```

Figura 91: Código de bloque de resaltado de caras

El bloque es muy similar al anterior, pero como se puede observar en la figura, además de contar la cantidad de objetos encontrados, genera una copia de la imagen y crea rectángulos alrededor de los objetos encontrados, con el fin de remarcarlos. A continuación, se muestra una imagen que ha sido enviada, junto con el resultado obtenido:

Figura 92: Imagen origen y resultado del resaltado de caras

Capítulo 6: Conclusiones

El objetivo principal planteado en la propuesta y además mencionado en el presente trabajo era el de obtener un framework que permitiera el procesamiento de datos en tiempo real, a través de la generación de diagramas de bloques funcionales.

Para cumplir con el mismo, se ha desarrollado primero una etapa de investigación, con el fin de evaluar las herramientas similares existentes, analizando las características positivas y negativas de las mismas, en conjunto con la investigación de algunos tópicos relacionados con el framework, tales como la programación visual y diagramas de bloques. El propósito de poder tener claros estos conceptos y las herramientas presentes, fue el de poder generar un instrumento nuevo que combine sus ventajas y que a la vez facilite su uso.

Como se ha indicado, estas herramientas por lo general son de simulación, más que de generación de sistemas reales, y funcionan en entornos aislados o dedicados. Por lo tanto, se propuso una herramienta en la cuales las dependencias sean mínimas y que pueda ser embebida dentro de un sistema externo para que su integración permita alterar la lógica y el flujo de datos de forma real, y no que solo se puedan realizar simulaciones que luego deben ser implementadas.

Luego se han listado una serie de características requeridas para el framework, que permitieran que el uso del mismo sea cómodo y eficaz, y, además, se ha desarrollado brevemente qué es un framework y cuáles son sus características necesarias, de modo de asegurar que el resultado del desarrollo cumpla con todas las condiciones necesarias para que realmente pueda ser clasificado como eso, un framework.

Una vez que el objetivo fue claramente especificado y luego de haber llevado a cabo el desarrollo del marco teórico relacionado con el trabajo, se ha desarrollado el framework propuesto, siempre teniendo en cuenta los requisitos fundamentales y los conceptos del marco teórico.

Concluido el desarrollo del software, se expuso en el Capítulo 3 la documentación del mismo, abarcando diferentes tópicos, iniciando con los requerimientos de software y la instalación, continuando con los primeros pasos para la integración y configuración para que sea utilizado en conjunto con un sistema externo y luego con el desarrollo de algunos conceptos inherentes del framework como los posibles tipos de ciclos, y los diferentes modos de ejecución. Además, también se listaron los diferentes bloques que el framework provee por defecto, especificando la funcionalidad particular de cada uno en conjunto con sus entradas. Para finalizar, se describió cómo extender el framework a través de la generación de nuevos bloques con nuevas funcionalidades, así como la generación de backends que permiten que el framework se acomode mejor a los diferentes requerimientos específicos de cada proyecto.

La exposición del capítulo 3, además, se acompañó con capturas de pantalla de una aplicación web desarrollada con el objetivo de demostrar las diferentes capacidades del framework, lo que además dejó en claro que el mismo puede ser embebido en cualquier sistema de forma sencilla,

y sin la necesidad de requerir de un software o plataforma específico para que pueda ser utilizado o ejecutado. En particular, este sistema web, podría ser mejorado para que funcione de manera responsiva en los navegadores de los dispositivos móviles, haciendo que la interfaz que conecta la aplicación con el framework pueda ser accedida y utilizada incluso de forma remota y móvil, lo que podría resultar en una herramienta potente y útil para muchos casos de uso.

El capítulo 4 fue llevado a cabo con el objetivo de mostrar algunas técnicas y conceptos informáticos utilizados para el desarrollo del framework, que podrían resultar de interés para entender en base a que conceptos el mismo fue desarrollado y para explicar el porqué de algunos mecanismos empleados para el uso del framework.

Para mostrar que el objetivo del trabajo fue alcanzado de forma satisfactoria, se han desarrollado en el capítulo 5 algunos casos de uso que permiten mostrar la utilidad del framework en los primeros ejemplos, así como también la sencillez a la hora de la generación de nuevas funcionalidades en los últimos.

Como conclusión final, se puede decir que la herramienta desarrollada cumple con los objetivos y características propuestas, así como también con todos los aspectos que debe cumplir un framework, y la efectividad del mismo ha quedado demostrado ya que se lo utilizó para el desarrollo de una aplicación web de ejemplo, y también para llevar a cabo la resolución de diferentes ejemplos/problemas planteados, de los cuales algunos se desarrollaron solo con los bloques provistos por el framework, y otros mediante la extensión del mismo, a través de la generación de nuevos bloques.

Para concluir con el trabajo, se presentará en el capítulo 7 una serie de posibles cambios/desarrollos aplicables al framework, donde los mismos se plantean con el objetivo de mejorar la eficiencia, la facilidad de uso, configuración e integración, la extensión de las prestaciones posibles, y la mejora en términos de escalabilidad y procesamiento distribuido.

Capítulo 7: Trabajos Futuros

7.1 Introducción

En este último capítulo se proponen diferentes modificaciones y/o extensiones que se podrían realizar sobre el framework con el fin de obtener una herramienta más completa y con mayores utilidades, y por lo tanto, más potente. Se presentan brevemente cada una de las mejoras planteadas a modo de mencionarlas, ya que la implementación y el desarrollo de cada solución acarrearía una extensa explicación.

7.2 Flowgramming como servicio externo

Un interesante trabajo que se podría realizar sobre Flowgramming es que el mismo pueda funcionar como un servicio externo a una aplicación de software en particular, y que pueda trabajar de forma aislada, pero comunicándose a través de algún tipo de interfaz. De esta forma, la instalación de Flowgramming se haría directamente sobre el sistema operativo, al igual que cual otro servicio/aplicación, y dispondría de una serie de directorios específicos y conocidos en los que se podrían incluir archivos de configuración, y en los cuales definir el código para nuevos bloques, campos, backends, etc.

Esta idea tiene en principio al menos tres ventajas. La primera y más relevante es que las aplicaciones que hacen uso de Flowgramming, no necesariamente van a tener que estar escritas en el lenguaje de programación que la herramienta (en este caso Python). Esto permitiría que el framework pueda ser utilizado por aplicaciones desarrolladas en cualquier lenguaje, siempre que éstas sepan cómo comunicarse con el servicio. Como restricción, la extensión de Flowgramming a través de generación de nuevas clases de bloques, campos, backends, etc. deberá seguir siendo bajo el mismo lenguaje de programación.

Como segunda ventaja, al poseer Flowgramming como un servicio externo, podría permitirse a más de una aplicación hacer uso de la misma instancia de Flowgramming. Si bien en un principio quizás no suene muy útil diseñar o editar diagramas a través de diferentes aplicaciones al mismo tiempo, si podría ser de mayor utilidad que diferentes aplicaciones puedan enviar datos a Flowgramming para que procese los datos en sus diferentes diagramas.

Como última, y quizás menos relevante ventaja, la instalación del framework podría resultar un poco más sencilla, ya que por lo general las herramientas que proveen los sistemas operativos tienen como resultado que la instalación de nuevos servicios sea más sencilla que la instalación manual que debe realizar un desarrollador para incluir una nueva herramienta en sus aplicaciones.

7.3 Migraciones de modificaciones y eliminación de tipos de bloques y campos

Actualmente, si luego de persistir un diagrama se elimina o modifica el código estructural de las clases de los bloques o las clases de los campos de los bloques que un diagrama persistido utiliza (es decir, el código que define la estructura de los bloques y campos, tales como el nombre de la clase y los atributos que la componen, sin incluir las modificaciones en el código que solo alteran el funcionamiento sin alterar la estructura), el framework queda en un estado de inconsistencia debido a que, al intentar levantar un diagrama, no puede encontrar las clases de los bloques/campos que utiliza.

Sería interesante poder definir algún método que permita crear migraciones de clases de forma automática y manual para tratar estas modificaciones. Esto es, que luego de modificar clases que definen bloques y campos, se pueda proveer una forma de alterar, actualizar o invalidar los diagramas que hacían uso de los mismos. El método funcionaría de manera similar a las migraciones de bases de datos que proveen muchos de los ORM actuales, donde automáticamente crean las migraciones a efectuar en la base de datos cuando se altera un modelo.

De esta forma, al alterar el código estructural de las clases, las migraciones permitirían actualizar los bloques diagramas de forma automática, sin dejar el framework en un estado de inconsistencia.

7.4 Evaluación de ejecución de bloques a través de una tabla

Como se ha detallado en la sección 4.4.4, cuando llega un nuevo dato al framework, la ejecución de los diagramas se realiza de forma recursiva, donde se buscan primero los bloques iniciadores correspondientes a la clave del dato recibido, y, recursivamente, se siguen todas sus relaciones para ir enviando las salidas de cada bloque, a los bloques que están conectados al mismo.

Este enfoque tiene algunas limitaciones. En primer lugar, se encuentra un problema de escalabilidad (tema que se abordará en la siguiente sección), ya que si se quisiera ejecutar el framework en varios procesos o hilos a la vez, cuando un proceso comienza a ejecutar un diagrama, el mismo se encargará de la ejecución completa del diagrama siguiendo las relaciones de los bloques, y si hubiera otros procesos disponibles y libres, no podrían ejecutar bloques del mismo diagrama de forma concurrente. Además, para casos de diagramas extremadamente complejos donde existe una gran cantidad de iteraciones, se podrían alcanzar niveles máximos de recursión permitidos, lanzando una excepción.

Como alternativa a esta metodología, se podría desarrollar un método más escalable y extensible, y que es una alternativa utilizada por algunas de las herramientas mencionadas en el Capítulo 2. La idea consiste en tener una tabla o una estructura similar en la que se pueda tener

un registro de todos los bloques para todos los diagramas persistidos, en conjunto con los estados de las entradas de cada uno, de la siguiente manera:

Id diagrama	Id bloque	Entrada 1	Entrada 2	(…)
Diagrama_1	Bloq_1	10	-	(…)
Diagrama_1	Bloq_2	5	20	(…)
Diagrama_2	Bloq_n	-	20	(…)
(…)	(…)	(…)	(…)	(…)

De esta forma, cuando llega un dato al framework, éste actualizará las entradas de cada uno de los bloques de entrada de los diagramas, y luego constantemente evaluará y ejecutará cada uno de los bloques que posean todas sus entradas listas para ser ejecutados. A la vez, luego de que cada bloque sea ejecutado, se actualizarán las entradas de los bloques relacionados con este para que estos puedan ser ejecutados posteriormente.

Para llevar a cabo la implementación de esta metodología, diferentes detalles tendrán que ser tenidos en cuenta, entre los más importantes:

- ✓ Cómo manejar los diferentes modos de ejecución (memorable y consumidor)
- ✓ Cómo identificar y frenar la ocurrencia de ciclos infinitos, o quizás no permitir la persistencia de diagramas cíclicos
- ✓ Cómo manejar el orden de ejecución de los bloques, o si es un detalle que no importa

La siguiente sección aborda el tema de multi-threading y cómo el presente esquema podría ser de gran utilidad para llevar a cabo la implementación de la solución.

7.5 Escalabilidad y paralelismo

La implementación de la solución presentada en la sección 7.4 trae aparejada, entre otras ventajas, diferentes facilidades que permitirán que el framework sea más escalable.

El hecho de poseer una tabla en la que se pueda evaluar constantemente si cada uno de los bloques puede ser ejecutados, deriva en una gran ventaja: en vez de tener un solo proceso encargado de ejecutar los diferentes bloques de un diagrama y seguir, de forma recursiva sus relaciones, se podría tener un grupo de “workers” en diferentes hilos de procesos, que de forma constante se encuentren buscando cuales son los bloques que pueden ser ejecutados en un momento dado, ejecutando aquellos que están listos, y realizando todas las operaciones derivadas de esa ejecución, como por ejemplo la actualización de las entradas de cada uno de los bloques ejecutados según el método de ejecución asignado, y de las entradas de los bloques que están enlazados al mismo.

Además, se podría llevar una instancia del framework a un modelo de ejecución distribuido, donde diferentes procesadores corran de forma concurrente diferentes workers, compartiendo entre todos una misma instancia de la tabla de bloques y entradas.

Si bien esta solución provee ventajas notables, es necesario aclarar que su implementación resulta algo compleja. Tanto en la solución distribuida como en la solución multi-threading monoprocesador, lo más relevante es, quizás, las consideraciones necesarias a tener en cuenta en cualquier solución concurrente. Por ejemplo, muy probablemente se requeriría de manejar bloqueos a nivel bloque, asegurar que en un momento dado no se esté ejecutando el mismo bloque en más de un proceso/hilo, de la misma forma que se debería bloquear la actualización de las entradas de cada uno de los bloques para que procesos no puedan actualizarlas al mismo tiempo.

7.6 Desarrollo y extensión dirigida por tests automatizados

Hoy en día, la tendencia en el desarrollo de software es que el mismo sea guiado por tests o pruebas. Existen diferentes metodologías, entre las que resalta TDD (Test Driven Development). Esta metodología, si bien posee elevados requisitos iniciales, a fin de cuentas permiten producir aplicaciones de mayor calidad y en menos tiempo. Permite que un programador se centre en la tarea específica y la primera meta es, a menudo, hacer que su desarrollo pase la prueba. Cuando la metodología es utilizada correctamente, se asegura de que todo el código escrito está cubierto por una prueba. Esto puede dar al programador un mayor nivel de confianza en el código.

En lo que respecta a Flowgramming, sería muy interesante poseer de un conjunto de tests automatizados que validen todo aquello que el desarrollador es capaz de extender en el framework, y que estas pruebas se ejecuten cuando el framework es instanciado al comienzo, lanzando excepciones en caso de que las pruebas no pasen. Como ventaja principal, el desarrollador podría depurar su código mientras lo va desarrollando, lo que conllevaría a una notable reducción de errores en tiempo de ejecución.

Aplicando TDD al framework, algunas de las pruebas que se podrían realizar son:

- ✓ Evaluar los bloques con diferentes datos para los tipos de entrada definidos, enviando valores “extremos” (por ejemplo, números muy grandes, muy chicos, nulos (0) y negativos para tipos numéricos, o vacíos y espacios para cadenas de caracteres, verificando que no haya errores en la ejecución.
- ✓ Validar que la salida de un bloque definido para los datos de prueba enviados se corresponde con el tipo de salida especificada para el bloque.
- ✓ Para los backends, verificar que las operaciones pueden ser llevadas a cabo con éxito y de la forma esperada. Por ejemplo, para backends de persistencia, almacenar un diagrama y al recuperarlo, verificar que el modelo es exactamente igual al almacenado.

- ✓ Que cada uno de los atributos definidos para los campos de una clase sean consistentes con el resto de los atributos definidos por la misma.

7.7 Mejoras de Serialización

Actualmente, por una decisión de simpleza, cuando se almacenan (persisten) los bloques de los diagramas, los mismos son serializados a través del módulo "pickle" de Python. El mismo permite que la mayor parte de los objetos Python puedan ser serializados, definiendo cómo convertirlos en datos de tipo textual, y como des-serializarlos.

El módulo pickle resulta muy cómodo y es de conocimiento para muchos programadores familiarizados con Python. Sin embargo, al comparar esta solución frente a otras soluciones, tales como serializar las instancias en objetos JSON, la original resulta ser pobre en términos de performance. Tanto la lectura como la escritura de objetos desde y hacia JSON, resulta ser mucho más eficiente cuando es comparada con "pickle". Solo a modo de mostrar a grandes rasgos la mejora de eficiencia, un estudio de velocidad que compara ambos métodos [44], revela que JSON es hasta 25 veces más rápido para de-serializar objetos y hasta 15 veces más rápido para serializarlos.

Por supuesto que el módulo "pickle" brinda muchas facilidades que permiten abstraernos de muchos detalles, por ejemplo, al convertir un objeto, ya sabe la localización del módulo de la clase que lo define, etc., mientras que JSON en un principio solo permite serializar tipos compatibles con el formato (listas, diccionarios, etc.), y la conversión a objetos sería un poco más compleja. Sin embargo, la notable diferencia de eficiencia del método JSON hace válido el esfuerzo requerido para su implementación, y además ya existen diferentes herramientas para tales fines, tales como "jsonpickle" [45].

Además, sería interesante proveer una forma en la que se le pueda permitir a los distintos desarrolladores la posibilidad de definir sus propios métodos de serialización, para que implemente una solución mejor si fuera necesaria.

7.8 Señalización de eventos

Podría ser de utilidad que los desarrolladores se puedan subscribir métodos o "callbacks" frente a la ocurrencia de cierto tipo específico de eventos conocidos. Es decir, proveer la utilidad de poder ejecutar un código específico cuando ocurre un evento en particular,

Por ejemplo, para el caso de análisis de imágenes de la sección 5.6, podrían definirse métodos que automáticamente creen y eliminen directorios para almacenar de forma temporal las imágenes procesadas o en procesamiento cuando el diagrama es persistido o eliminado respectivamente.

Entre los eventos interesantes a los que se podría permitir que el desarrollador asigne funciones a ejecutar, podrían listarse:

- ✓ La persistencia de un diagrama en particular
- ✓ La eliminación de un diagrama en particular
- ✓ El comienzo de la ejecución de un bloque
- ✓ El fin de la ejecución de un bloque
- ✓ El envío de la salida de un bloque a la entrada de otro bloque

7.9 Migración de un backend de persistencia a otro

Podría suceder que para una instancia de Flowgramming que se encuentra en utilización, se decida alterar o modificar el backend de persistencia que posee. Una utilidad interesante sería la de proveer una funcionalidad que permita obtener el estado actual de los diagramas y los bloques almacenados para la configuración de un backend en particular y poder copiar todos esos modelos en el formato que un nuevo backend defina, junto con su configuración específica.

De esta forma, los desarrolladores podrían alternar, cambiar o probar diferentes backends de persistencia manteniendo las instancias de los modelos que ya se encuentran persistidos.

7.10 Modos de ejecución a nivel bloque

Los modos de ejecución memorable y consumidor ya definidos y explicados en la sección 3.5 son aplicables a cada diagrama cuando los mismos son persistidos. Sin embargo, podría ser de utilidad en algunos casos que estos modos puedan ser aplicados para cada bloque de manera particular.

Para llevar a cabo una implementación que cubra esta característica, hay algunos detalles que no pueden dejarse de lado. Por empezar, hay que modificar la forma actual en la que los diagramas son almacenados o definidos para que el modo de ejecución pueda ser detallado a nivel bloque, es decir, con una granularidad más pequeña. Además, este aspecto debe ser tenido en cuenta en tiempo de ejecución, en el momento en que cada bloque es ejecutado, para definir qué es lo que se debe realizar con los valores de sus entradas. Por último, un detalle importante es el de cómo validar los bloques en términos de presencia de ciclos, siendo necesario volver a pensar las reglas para identificar ciclos, que fueron detalladas en la sección 3.7.

7.11 Modo de pruebas aislado

Actualmente, cuando un usuario final de la aplicación que hace uso del framework se encuentra desarrollando/editando un diagrama, este puede enviar datos de prueba como se explicó en la

sección 3.8.3, donde con la ayuda de los backends de debugging, puede ver cómo el diagrama procesa los datos de prueba enviados en tiempo real.

Al enviar los datos en modo testing, lo que el framework hace es ejecutar los diagramas para las claves indicadas, a su vez notificando a los backends de debugging, que de otra forma se encuentran inactivos. Sin embargo, si se envía un dato para una clave, por ejemplo “x” que otro diagrama también utiliza (además del que el usuario se encuentra editando), este diagrama también es ejecutado.

Sería útil que al enviar un dato en modo testing, solo se le envíe este dato al diagrama que está siendo utilizado. Aunque si se quisiera ver cómo el diagrama procesa los datos en interacción con otros diagramas, esto no sería posible. Por lo tanto, esta funcionalidad debería proveerse como opcional a la hora de enviar datos de prueba.

7.12 Modo de indicar tipos de entrada y salida de un bloque y sus campos

La forma de indicar los tipos de salida y de los bloques y los tipos de entrada de cada uno de los campos es simplemente citando los tipos de las clases de Python, como ya se ha demostrado anteriormente.

Quizás debería evaluarse una forma más prolija de indicar los tipos, a través de algún método más flexible, que permita generar una herencia de tipos en la que se puedan aceptar diferentes tipos como entrada, por ejemplo, y definir tipos propios, así como indicar tipos que sean compatibles.

Sin embargo, en caso de desarrollarse una solución, habría que evaluarla detalladamente en conjunto con su complejidad, ya que la idea es la de brindar flexibilidad y no la de generar una metodología que solamente conlleve a una implementación más compleja, o a un uso más complejo del framework.

7.13 Mejoras Menores

A continuación, se listan tres mejoras que implicarían cambios relativamente menores, pero que podrían brindar algunas utilidades adicionales:

1. **Identificación de ciclos infinitos de forma temprana:** sería útil y aportaría una mejora de eficiencia un método un poco más inteligente que permita la identificación temprana de ciclos infinitos en tiempo de ejecución, sin tener que esperar a que la ejecución genere demasiadas iteraciones sobre el ciclo. Sin embargo, de llevarse a cabo una solución, debe verificarse que la misma provea una mejora de eficiencia y no que su implementación agregue mucha más carga en tiempo de ejecución. Además, podría evitarse esta

verificación cuando se ejecutan bloques pertenecientes a diagramas que no presentan ningún tipo de ciclo identificado.

2. **Backends de logging de errores:** Actualmente, los backends de notificación de errores solo notifican errores que ocurren durante la ejecución de un diagrama. Podría ser útil que además notifiquen la ocurrencia de cualquier error o excepción ocurrida durante cualquier fase de la utilización de Flowgramming, como por ejemplo durante la persistencia de un diagrama.
3. **Deshabilitar bloques base:** para algunas aplicaciones o para algunos fines, algunos o muchos de los bloques provistos por Flowgramming podrían no ser útiles. Por lo tanto sería interesante poseer una forma sencilla de ocultar o deshabilitar los mismos mediante la configuración de una forma sencilla y sin necesidad de editar el código fuente.

Tabla de figuras

Figura 1: Diagrama de bloques de una CPU 10
Figura 2: entorno de Scratch..... 13
Figura 3:esquema de un VPL inspirado en diagramas de flujo..... 14
Figura 4: Programa ejemplo de Flowgorithm..... 14
Figura 5: programa de flujo de datos 15
Figura 6: Programa realizado con un VPL de máquina de estado finito..... 16
Figura 7: árbol de comportamiento..... 17
Figura 8: reglas basadas en eventos 18
Figura 9: pipe de procesamiento de flujo de datos 21
Figura 10: estructura de un bloque de diagrama de bloques funcionales 23
Figura 11: Diagrama de bloques funcionales..... 24
Figura 12: Ciclo de ejecución de un programa en un PLC..... 25
Figura 13: Programa de ejemplo de un PLC..... 26
Figura 14: Programa ejemplo de Microsoft VPL 27
Figura 15: Inversión de control 28
Figura 16: Código ejemplo de Flask #1 29
Figura 17: Código ejemplo de Flask #2 30
Figura 18: Arquitectura de un framework 33
Figura 19: Framework instanciado 33
Figura 20: Aplicación de ejemplo de uso de Flowgramming 36
Figura 21: estructura sugerida para el uso de Flowgramming Framework dentro de un proyecto 38
Figura 22: Código básico necesario para instanciar Flowgramming 38
Figura 23: Código de obtención de los diferentes tipos de bloques para construir un panel 39
Figura 24: Resultado (reducido) del método 'get_blocks_data_for_panel'..... 40
Figura 25:Instancia de un bloque de Suma (Add)..... 41
Figura 26: Representación de un bloque de entrada de punto flotante..... 42
Figura 27: Diagrama sencillo de ejemplo..... 43
Figura 28: Ejemplo de persistencia de un diagrama en Flowgramming..... 45
Figura 29: Obtención de datos de diagramas persistidos 47
Figura 30: Salida de ejemplo de datos de diagramas persistidos..... 47
Figura 31: Tabla de diagramas persistidos..... 48
Figura 32: Código de obtención de un diagrama..... 48
Figura 33:Ejemplo de estructura devuelta por Flowgramming para un diagrama 49
Figura 34: Código de ejemplo de envío de datos a Flowgramming 50
Figura 35: Ejemplo de flujo de datos en un enlace..... 51
Figura 36:: Ejemplo de flujo de datos #2 51
Figura 37: Ejemplo de flujo de datos en modo Memorable..... 52
Figura 38: Ejemplo de flujo de datos en modo Memorable #1..... 52

Figura 39: Ejemplo de flujo de datos en modo Memorable #2 53
Figura 40: Ejemplo de flujo de datos en modo Memorable #3 53
Figura 41: Ejemplo de flujo de datos en modo Memorable #4 53
Figura 42: Persistencia de un diagrama en diferentes modos de ejecución 54
Figura 43: Ejemplo de ejecución condicional: estado #1 55
Figura 44: Ejemplo de ejecución condicional: estado #2 56
Figura 45: Ejemplo de ejecución condicional: estado #3 56
Figura 46:: Ejemplo de ejecución condicional: estado #4 57
Figura 47: Ejemplo de diagrama con ciclos.. 58
Figura 48: Ejemplo de ciclo crítico 59
Figura 49: Ejemplo de ciclo no crítico 60
Figura 50: Conversión de ciclo crítico en no crítico 60
Figura 51: Ejemplo de resultado de verificación de un diagrama con errores............................ 61
Figura 52: Ejemplo de instanciación de Flowgramming con backend de persistencia específico 64
Figura 53: Ejemplo de instanciación de Flowgramming con backend de persistencia "personalizado" 64
Figura 54: Envío y notificación de datos de prueba... 65
Figura 55: Ejemplo de instanciación de Flowgramming con backend de logging de errores específico 67
Figura 56: Código del bloque de suma 72
Figura 57: Definición de bloque personalizado. Paso #1 74
Figura 58: Definición de bloque personalizado. Paso #2 74
Figura 59: Definición de bloque personalizado. Paso #3 74
Figura 60: Definición de bloque personalizado. Paso #4 75
Figura 61: Ejemplo de instanciación de Flowgramming con rutas donde buscar bloques extra . 75
Figura 62:Utilización de bloque personalizado.. 76
Figura 63: Salida compleja de un bloque 76
Figura 64: Formateo de la salida de un bloque 76
Figura 65: Salida formateada de un bloque.. 77
Figura 66: Diagrama con casting de datos 78
Figura 67: Uso del parámetro la opción de fields "default_frontened_value" 79
Figura 68: Uso del parámetro la opción de fields "is_input".. 79
Figura 69: Uso del parámetro la opción de fields "disable_default_value" 80
Figura 70: Uso del parámetro la opción de fields "disable_connector"...................................... 80
Figura 71: Código de ejemplo de un campo personalizado 81
Figura 72: Bloque de envío de resultados 83
Figura 73: Uso de bloque de envío de resultados en diferentes etapas de un diagrama 83
Figura 74: Bloque de división segura 84
Figura 75: Ejemplo de uso de persistencia post-ejecución 85
Figura 76: Implementación de un backend de Debugging 89
Figura 77: Implementación de un backend de notificación de errores...................................... 90

Figura 78: Clase Declarativa 91
Figura 79: Ejemplo de especificación de rutas relativas para bloques y backends 93
Figura 80: Diagrama del caso de ejemplo #1 97
Figura 81: Salida de la solución del caso de ejemplo #1 97
Figura 82: Diagrama del caso de ejemplo #2 A 98
Figura 83: Salida de la solución del caso de ejemplo #2 A 99
Figura 84: Diagrama del caso de ejemplo #2 B 99
Figura 85: Salida de la solución del caso de ejemplo #2 B 100
Figura 86: Diagrama del caso de ejemplo #3 101
Figura 87: Salida de la solución del caso de ejemplo #3 101
Figura 88: Diagrama de cinta de producción industrial 102
Figura 89: Bloque de conteo de caras 104
Figura 90: Diagrama de conteo de caras en imagen 104
Figura 91: Código de bloque de resaltado de caras 105
Figura 92: Imagen origen y resultado del resaltado de caras 106

Referencias bibliográficas

1. http://www.controleng.com/single-article/function-block-diagrams/ca572fcc91f7fe98f033fe57cff822b2.html
2. https://www.mathworks.com/help/simulink/
3. Mohammad Nuruzzaman (2004), Modeling and Simulation In SIMULINK for Engineers and Scientists (ISBN: 9781418493837)
4. https://www.scilab.org/scilab/features/xcos
5. http://wolfram.com/system-modeler/features/
6. http://www.ni.com/white-paper/53642/es/
7. https://www.openmodelica.org/
8. https://en.wikipedia.org/wiki/Modelica
9. https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_componentes
10. http://www.eurekamagazine.co.uk/design-engineering-features/technology/acausal-modelling-sees-problems/11500/
11. https://en.wikipedia.org/wiki/Web-based_simulation
12. https://www.smartdraw.com/block-diagram/
13. http://www.craft.ai/blog/the-maturity-of-visual-programming/
14. https://en.wikipedia.org/wiki/Function_block_diagram
15. http://www.carldyke.com/function-block-diagrams
16. https://booksite.elsevier.com/9781856176217/appendices/01~Ch11.pdf
17. W. Bolton (2009), Programmable Logic Controllers Paperback (ISBN: 9780750681124)
18. https://www.techtransfer.com/blog/basics-plc-operation/
19. Programming with STEP 7 Manual - Siemmens
20. https://www.slideshare.net/saintkepha/autonomic-computing-dataflow-programming-and-reactive-state-machines
21. Johnston, Wesley M.; J.R. Paul Hanna; Richard J. Millar (2004), Advances in Dataflow Programming Languages
22. https://en.wikipedia.org/wiki/Dataflow_programming
23. https://en.wikipedia.org/wiki/Stream_processing
24. https://en.wikipedia.org/wiki/Reactive_programming
25. https://msdn.microsoft.com/en-us/library/bb483088.aspx
26. http://www.cs.ucf.edu/~dcm/Teaching/COT4810-Spring2011/Literature/DataFlowProgrammingLanguages.pdf
27. Johnson, R E (1992), Documenting frameworks using patterns
28. http://www.thefreedictionary.com/framework
29. http://flask.pocoo.org/docs/0.12/
30. https://info.cimetrix.com/blog/bid/22339/what-is-a-software-framework-and-why-should-you-like-em
31. https://en.wikipedia.org/wiki/Software_framework
32. http://www.thefreedictionary.com/framework
33. https://www.reddit.com/r/explainlikeimfive/comments/4983hi/eli5_framework_what_is_it_in_terms_of_computer/
34. Trudeau, Richard J. (1993). Dover Pub., ed. Introduction to Graph Theory (ISBN: 9780486678702)

35. https://es.wikipedia.org/wiki/Grafo
36. http://minhhh.github.io/posts/a-guide-to-pythons-magic-methods
37. http://mqtt.org/
38. https://www.eclipse.org/paho/
39. https://www.sqlalchemy.org/
40. https://docs.djangoproject.com/en/1.11/topics/db/
41. https://scrapy.org/
42. https://django-haystack.readthedocs.io/en/master/
43. https://github.com/j4mie/micromodels
44. https://konstantin.blog/2010/pickle-vs-json-which-is-faster/
45. https://jsonpickle.github.io/
46. https://opencv.org/
47. https://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html
48. https://virtualenv.pypa.io/en/stable/
49. https://www.w3schools.com/js/js_json_intro.asp